ISSN 0316-6295

# Pted: a Standard Pascal text editor based on the Kernighan & Plauger design

by Ken Newman
Department of Computer Science
University of Toronto

Technical Report CSRG-120
October 1980

COMPUTER SYSTEMS RESEARCH GROUP
UNIVERSITY OF TORONTO

PB1-0046-(05)

# Pted: a Standard Pascal text editor based on the Kernighan & Plauger design

by Ken Newman
Department of Computer Science
University of Toronto

Technical Report CSRG-120
October 1980

Computer Systems Research Group
University of Toronto
Toronto, Canada
M5S 1A1

The Computer Systems Research Group (CSRG) is an interdisciplinary group formed to conduct research and development relevant to computer systems and their application. It is jointly administered by the Department of Electrical Engineering and the Department of Computer Science of the University of Toronto, and is supported in part by the Natural Sciences and Engineering Research Council of Canada.

ABSTRACT

Pted is a portable UNIX-style text editor written entirely in PT, a fully compatible subset of Standard Pascal, as defined in (Jensen & Wirth74). PT has no function procedures and only sequential files. Rosselet has developed a portable, self-compiling PT compiler, which presently generates code for the PDP-11 (Rosselet 1980). Pted is based on the editor of Kernighan and Plauger given in (Kernighan & Plager76), which is virtually identical in command syntax and philosophy to the UNIX editor "ed" written by D.M.Ritchie and K.L. Thompson at Bell labs; there are, however, differences that will be detailed later on. Pted was written in response to a need for a good machine-independent editor written in a nicely structured high-level language.

Copyright (C) 1980 by the University of Toronto

Digitized by the Internet Archive
in 2018 with funding from
University of Toronto

https://archive.org/details/technicalreportc120univ

Contents


| | | |
|---|---|---|
| | Acknowledgements | 3 |
| 1. | Brief Beginner's Guide to Pted. | 4 |
| 2. | Similarities to UNIX editor "ed"; Formal Feature Summary. | 5 |
| 3. | Differences from UNIX editor "ed". | 8 |
| 4. | Installing Pted; some problems and example solutions on Unix. | 10 |
| 5. | Modifying Pted; implementation details, worthwhile extensions, profile results. | 13 |
| 6. | Pted's size, capacity limitations, hardware comments. | 15 |
| | Appendix 1: Pted source. | 17 |
| | Appendix 2: C shell program used on Unix. | 66 |
| | Appendix 3: Test script. | 72 |
| | Bibliography | 78 |

ACKNOWLEDGEMENTS

I wish to express my gratitude to B.W. Kernighan and P.J. Plauger for their excellent (almost flawless) design, which motivated this ambitious but rewarding project. If the many other tools they present in (Kernighan & Plager76) are of the quality of their editor, as it seems they are, theirs is one book that is an essential part of every programmer's library.

Thanks also to Professor R.C. Holt for suggesting and supporting this project, and for his advice and criticisms, and to Alan Rosselet, who in addition to writing the PT compiler gave valuable help with the judicious use of the flexibility of the UNIX operating system.

Thanks also to K. Thompson and D. Ritchie for the remarkable operating system, Unix, which they developed at Bell Labs. The references to Unix in what follows refer to their systems.

## section 1: Brief Beginner's Guide to Pted

This section will give a summary of the philosophy of pted, rather than give a tutorial on its use, which is available from other sources ([Kernighan & Plauger76], [Kernighan]); however, a concise formal summary of the features of pted will be given later for reference.

Pted is an interactive line-oriented text editor streamlined for conversational use, i.e. its commands and responses to them are generally very brief and to the point. Pted does not edit files "in place"; they are copied to scratch files, which are then manipulated in response to user commands. No further reading or writing of files is done automatically unless explicitly requested; in particular no automatic backup is written, but must be explicitly written before leaving the editor. This allows one to more easily edit a file just for the purpose of reading it, when you have no intention to change it, as well as preserving the original file in case an edit is messed up. The user need have no knowledge of the underlying buffering or storage of text, but can view the text as a single unit of varying size.

The text is viewed as a single stream of characters with embedded "newline" characters (equivalent to the combination "CR-LF" on some systems) that break it up into lines. There is thus no set line length; neither is there any explicit numbering of lines, since all line numbers are relative to the beginning of the buffer. All editing commands consist of a single letter, optionally preceded by one or two line number specifiers, which would override a default if present. The line number specifier(s) indicate the range of lines the command is to act on. The current line and last line have special status, and can be selected by metacharacters. Pted also contains very extensive error checking in order to prevent crashing and loss of the updates made to the original file, and indicates errors by a "?" followed by a single-letter code (listed later).

Perhaps the most powerful feature is the "context-search" ability combined with a flexible pattern-matching facility, which recognizes text patterns that are a subclass of the class of regular expressions. With this feature, searching for certain strings in the text becomes particularly convenient.

The conventions outlined above have been tested in use in a number of widely accepted conversational text editors and have been praised for their usefulness and human engineering. A formal summary of pted's features will be given next, for reference.

## section 2: Similarities to UNIX editor "ed"; Formal Feature Summary.

As mentioned above, pted is very similar to ed in both command syntax and philosophy. In particular, pted supports the features below (adapted from [Kernighan & Plauger76]). Items in square brackets are optional except when in the context of a character class where the square brackets are metacharacters; line numbers in parentheses are the default and the parentheses are not part of the command syntax. Some metacharacters were changed from the K & P assignments for local system compatibility.

Line numbers are formed from the following components:

| | |
|---|---|
| 17 | a decimal integer (i.e. 17) |
| . | the current line |
| $ | the last line |
| /.../ | a forward context search |
| ?...? | a backward context search |

These may be combined with + or -:

| | |
|---|---|
| .+1 | sum of . and 1 |
| $-7 | difference of $ and 7 |

Also, line numbers beginning with + or - are taken to mean relative to the current line ".". Line numbers are separated by commas or semicolons; a semicolon sets the current line to the most recent line number before proceeding.

Commands:

| | |
|---|---|
| (.) a | append text after line (text follows) |
| (.,.) c | change text (text follows) |
| (.,.) d[p] | delete text |
| (.) i | insert text before line (text follows) |
| (.,.) m line3[p] | move text to after line3 |
| (.,.) p | print text |
| q | quit |
| (.,.) s/pat/new/[g][p] | substitute new for first occurrence of pat (g implies repeatedly across line) |
| (.) =[p] | print line number |
| (.+1) | print next line |

and global prefixes

| | |
|---|---|
| (1,$) g/pattern/command | |
| (1,$) v/pattern/command | |

where "command" can't be a,c,i,q, or either of the global

prefixes g,v. The global prefixes cause repeated execution of "command", once for each line that matches (g) or does not match (v) "pattern"; dot is set to the matched line before "command" is done. There is also for convenience the special command "&" which, alone on a line, is equivalent to ".,.+18p" if there exist at least 18 lines after the current line, and to ".,$p" if not. If explicit line numbers are given without a command following them, the range of lines indicated is printed. The file I/O commands "r" and "w" will be explained in the next section.

The following is a summary of the features of the pattern matcher.

A text pattern consists of the following elements:

| | |
|---|---|
| c | literal character |
| . | any character except newline |
| ^ | beginning of line |
| $ | end of line (null string before newline) |
| [...] | character class (any one of these characters) |
| [^...] | negated character class (all but these characters) |
| * | closure (zero or more occurences of previous pattern) |
| \c | escaped character (e.g. \^, \[, *) |

Any special meaning of characters in a text pattern is lost when escaped, inside [...], or when the special meaning can't logically be applied, as in the following cases:

| | |
|---|---|
| ^ | not at beginning |
| $ | not at end |
| * | at beginning |

A character class consists of zero or more of the following elements, surrounded by [ and ]:

| | |
|---|---|
| c | literal character, including [ |
| a-c | range of characters (digits, lower or upper case) |
| ^ | negated character class if at beginning |
| \c | escaped character (\^, \-, \\, \]) |

Special meaning of characters in a character class is lost when escaped or when the special meaning can't logically be applied, as in the following cases:

| | |
|---|---|
| ^ | not at beginning |
| - | at beginning or end |

A substitution pattern consists of zero or more of the following elements:

| | |
|---|---|
| c | literal character |
| & | ditto, i.e., whatever was matched |
| \c | escaped character (\&) |

An escape sequence consists of the character \ followed by a single character:

| | |
|---|---|
| \n | newline |
| \t | tab |
| \c | c (including \\) |

The single-letter error message codes that can appear following a "?" are:

| | |
|---|---|
| f | file specifier syntax error |
| g | illegal command with global prefix |
| m | tried to move to an illegal place (e.g. 1,6m4) |
| o | tried to read a non-existent file |
| p | pattern syntax |
| s | no substitutions found |
| T | pted file system overflow |
| x | unknown command |
| $ | line address out of range |
| / | context search failed |

and there are two additional messages that indicate an internal inconsistency and should never happen:

| | |
|---|---|
| ?ps | unrecognizable encoded pattern entry |
| ?om | unrecognizable encoded pattern entry |

## section 3: Differences from UNIX editor "ed".

The major appearance differences are pted's much restricted file I/O commands. They were restricted to preserve compatibility with Standard Pascal, i.e. to make the program self-contained and operating system independent. In particular, no external file names may be explicitly typed within the editor; external files are passed to the program in the standard way, with the compiler associating the external file names with the internal file names in the program header when the editor is invoked. Files are thus known to the editor at compile time and no communication with the operating system is done. This has implications for the user interface that will be detailed shortly, along with the solution used on UNIX. The "e" and "f" commands of ed don't exist; the "r" and "w" commands are restricted to the forms below. The remarks below are in the context of the actions taken by pted on the internal files named in the program header.

| | |
|---|---|
| (.) r | read main input file "maininp", which must exist. |
| (1,$) w | write main input file "maininp", which must exist. |
| (.) r 1 | read first read file "r1", which must exist. |
| (.) r 2 | same as above for second read file "r2". |
| (1,$) w 1 | write on first write file "w1", which needn't exist. |
| (1,$) w 2 | same as above for second write file "w2". |

The effective meaning of the r and w commands can and should be changed from the above; how this can be done will be discussed shortly.

The remaining appearance differences are very minor, for example, the number emitted after file I/O operations in pted is a line count, not a character count as in "ed"; this gives a better intuitive feel for the size of a textfile. There are also some cases in patterns where a metacharacter cannot logically stand for anything but itself and hence needn't be escaped; pted is very consistent in this regard, not requiring an escape when the meaning is already unambiguous, whereas ed in some cases does and in some cases does not require an escape. Other appearance differences are: ed, often due to local modifications, has more commands and allows "a", "c", or "i" commands with a global prefix; ed often has additional pattern-matching constructs, allows UNIX commands to be executed from within itself (including nested editors), and has the facility to catch interrupts (for example, it writes out the buffer if a "hangup" is received, meaning the terminal connection has been broken). Most of these differences are due to ed's direct system calls from C, something that pted doesn't do, to remain operating-system independent.

The internal differences, i.e. implementation differences, are quite significant, however, mainly due to ed's use of random-access files and system buffering (primitives are UNIX system calls), and pted's use of sequential files with its own primitives. These internal differences and their performance implications (among others, longer response time on some operations for pted) will be discussed shortly.

## section 4: Installing Pted; some problems and example solutions on Unix.

One problem is primarily one of convenience relating to the user interface; pted has to know about 7 external files in addition to the standard files "input" and "output", and users would soon tire of supplying all these file arguments each time before running the editor. In most Pascal implementations the file arguments can't simply be omitted, either, since a default is usually then used for the file names, often simply the internal file names themselves. Furthermore, one usually cannot even concoct some imaginary names in the case of read files, since they must physically exist prior to being "reset". There are three main solutions to this problem (in order of increasing palatability):

- put up with the arguments.
- simplify the file I/O of pted to reduce the number of arguments.
- provide some sort of intermediate program that will be the interface between users and pted and will generate its own default file arguments for those not supplied by the user, as well as remove any so generated after the user quits, when necessary.

The last solution is the preferred one, and was used in the UNIX version - the C shell program included in the appendix accomplishes this. This might be possible only in assembler on other systems. The convenience features provided by the C program will now be outlined to better demonstrate the usefulness of such a program.

The syntax of the command invoking the UNIX version of pted (i.e. invoking the C shell program) is as follows, where the square brackets indicate optional items:

pted inputfile [-r r1 [r2] ] [-w w1 [w2] ]

where "inputfile" must be specified but needn't have existed prior to invoking the editor. Similarly, if "r1" and "r2" didn't previously exist, they are created. Attempts to read in a previously non-existent file will give a "?o" warning as in ed. You cannot, if one of r1,r2 is the same as one of w1,w2, write on the common file and then read it back in; the read command will not work properly (warning "?o" given). The "-r" flag indicates a maximum of two read files follow (the first, r1, would be read by (.) r 1 and the second, r2, by (.) r 2 ). The "-w" flag indicates a maximum of two write files follow (the first, w1, would be written by (1,$) w 1 and the second, w2, by (1,$) w 2 ). A write command cannot fail, whether the files (if any) specified previously existed or not.

The r and w commands take on an expanded meaning because of the C interface program; in particular the external file names the user supplies are remembered and defaults also supplied so when r or w commands are issued the editor can determine if they are sensible. The new meanings for the UNIX version r and w commands are as follows (to give the general idea):

| | |
|---|---|
| (.) r | read main input file (which must have been given); error if previously non-existent. |
| (1,$) w | write main input file (which must have been given); needn't have existed previously. |
| (.) r 1 | read first read file specified in command invoking pted; see above. Error if one not given. |
| (.) r 2 | same as above for second read file. |
| (1,$) w 1 | write on first write file specified in command invoking pted; see above. If one was not given, this command will take a safe default action and write on a file wi.xxxxx in the current working directory, where "xxxxx" is a unique process id number. The intention is that users not rely on this default action. |
| (1,$) w 2 | same as above for second write file; the corresponding default is w2.xxxxx . |

The key point here is that now these commands read or write whatever files the user supplied when pted was invoked, or give warnings when reading is attempted on files that weren't supplied (or were supplied but didn't exist).

A minor detail problem might arise relating to the convention used for "newline" characters in the text. On UNIX, newline is a dedicated character; in fact it replaces the conventional "line-feed" character in ASCII, although such conventions are user-assignable on an individual basis. The constant "newline" is assigned the value 10 (decimal) in pted; if on the target system "carriage return" serves as a newline, "newline" should be instead assigned 13. If however carriage-return and line-feed serve their original primitive functions, i.e. each line must end in "CR-LF" the fix is less trivial. In this case the much preferred action would be to arrange it so the monitor or shell allocates to one character this double function and assign that character's ASCII code to "newline". The alternative involves a minor amendment to 3 small procedures.

It is conceivable that on very small Pascal systems some difficulty might be encountered in compiling pted. Pted is a moderately large program, and nests quite deep; it also requires a fair number of global constants, types, and variables. If compiler stack or table capacities are exceeded, and they can't be increased, unfortunately non-trivial modifications to pted would be required, involving deleting less important features. If the problem is one of insufficient memory available for arrays, the capacity of pted can be reduced by decreasing some array size

constants; these should be easily locatable in the source listing in the Appendix.

## section 5: Modifying Pted; implementation details, worthwhile extensions, profile results.

The file I/O in pted had to be extensively redone from the Kernighan and Plauger design; in the latter, file I/O is left largely unspecified, relying on system calls. Two main problems become apparent when using Pascal as the source language: sequential files only and lack of capability to handle a true read-write file. Added to these is the lack of system call dependence; a "seek" primitive had to be written which seeks to a specified character offset in a scratch file. This procedure is where pted spends most of its time since, due to the variable length of lines of text, file I/O is all done at the single-character level; this includes seeks. In particular, forward seeks are done by reading characters forward from the current position, and backward seeks are done by resetting the file to the beginning and then reading characters forward from there. A very beneficial modification, if possible on some Pascal system, would be to replace this seek primitive with a system random-access primitive. Of course this would violate some portability considerations; Standard Pascal specifies sequential files only, and a reason Standard Pascal is so machine-independent is that this is a standard.

Another problem was caused by the lack of true read-write files. This problem was solved by having two scratch files instead of the normal single scratch file, and conceptually using one as a read and one as a write file. Of course, this wouldn't work that simply, since it is always required that you be able to read the latest text written. This then means that either: the read file must be changed from a read file to a write file (i.e. copied in its entirety to a second scratch file) before writing and the write scratch file must be reset for reading after writing; or the above is done but some text is buffered into memory to avoid file duplication etc. before every writing operation. The latter alternative was chosen. The behaviour of this system can be summarized as follows. The current scratch file is normally read-only, and when a command requests a write operation, a check is made to see if the line of text will fit in a fairly large in-core buffer. If it will, it is stored there and no file manipulating is done. If not, i.e. the in-core buffer is full, the current read-only scratch file is copied in its entirety to a second scratch file and the contents of the in-core buffer are appended at the end. This write-only file is then reset to become read-only and becomes the current scratch file, with the in-core buffer now empty again. The effect of the in-core text buffer is then to provide a read-write extension to the current read-only scratch file, postponing unavoidable file duplications. As a practical necessity, the generations of scratch files are passed back and forth between two physical files, only one being used at a time. In practice this technique is usually invisible; the real

slowdown is the seeking operation.

The attractive feature of the dual scratch-file technique is that it is accomplished with no change at all in the text line pointer strategies, i.e. as far as the pointers are concerned, the text is all in one big read-write file. This is accomplished by interpreting the pointer values with respect to the end of the current physical scratch file, i.e. determining "where to look" for the line, on the scratch file or in memory. There is one exception to the use of the scratch file/memory buffer operation; when starting up, i.e. copying the main input file to a scratch file, the in-core buffering is disabled for speed, so the text is almost literally just copied into the scratch file (except for the building of the pointer system as this proceeds). This has the advantage of speeding up the file copy but the disadvantage that the in-core buffer will be empty so initial accesses all go to the scratch file and will be slower than if the lines were in memory.

Pted modifications on a smaller scale than file I/O would be quite simple to make, given that the modifier is very familiar with [Kernighan & Plauger76]. Modularity was a prime consideration in the design and commands can be added, for example, with little trouble since the mechanisms for parsing off line numbers, trailing p's, patterns, etc. are all separate modules. Trivial changes like changing metacharacters etc. generally involve just changing a Pascal "const".

A profiling facility available on Unix was used to determine where pted was spending its time. (A profiler is a program that generates statement execution counts for a source program). The significant heavily-executed parts will be briefly discussed here. The section consistently generating the highest counts was, as discussed earlier, the procedure implementing the "seek" primitive for the sequential files. Also high was the procedure reading from the scratch files after the seek was done. Very high also, although quite fast, was the procedure that follows links to obtain line indexes; this technique starts at the first line and links through the entire list until the index is found. The primitives for reading and writing one line to and from a file were also fairly highly used. Another primitive that adds items to an array and does a subscript range check, because of its universal use, was moderately heavily used, as were the three or four top-level routines in the pattern-matcher. The remaining code was insignificant in terms of increasing response time. Of course the execution sequences depend strongly on what type of commands you are profiling, but the above should dominate profiles of an "average" statistically meaningful group of sessions.

## section 6: Pted's size, capacity limitations, hardware comments.

To give an idea of the size of the pted program, the size details on UNIX are: as a PT object file, 18,872 bytes; in standalone form as a UNIX load module (i.e. including the PT runtime monitor) 25,834 bytes. The source is 76,631 bytes, but this is paragraphed and commented with many blanks, tabs, etc.

The capacity of pted (i.e. the amount of text it can keep track of) depends on the memory the system allocates for it and the number of bits in a Pascal integer word. Pted is currently set up for 16-bit integers, which is usually the bare minimum, and will work fine with anything larger as well, but not, in its present form, with 8-bit Pascal integers. However if integers are 32-bit or larger on the target system the capacity may have to be decreased since the system may not be able to provide enough memory, even though the full word is not being used - if 16-bit integers are available, they should be used. Pted's capacity may be decreased by the judicious changing of some array size constants, which should be easily locatable in the listing in the Appendix.

The capacity limitations of the UNIX version of pted are:

- in its file system, a maximum of 64,000 characters or 1598 lines, whichever is smaller. The former is due to the integer range limit in a 16-bit signed word, the latter to the size of the array holding pointer blocks, which is 8000 (each line uses 5 words of pointer information).

- in the number of auxiliary files it can handle, its limits are 2 read files and 2 write files independent of the main input file.

- the maximum allowed line length is 120 characters.

- the maximum allowed encoded pattern length is 175.

- the size of the in-core text buffer mentioned earlier is 2000 characters.

The hardware necessary to physically run pted is probably quite minimal, although unfortunately this has not yet been tested. However small microprocessor systems might be unbearably slow due to the sequential file restriction; if recoding can be done for random access files this might allow smaller systems. Fast disk systems are preferred, although floppy disks or high-density tape drives are a possibility (best checked by empirical tests). Tape is useable theoretically with the design. Memory should be

at least 64K for running if the full-capacity version is desired; smaller sizes would make maintenance awkward (i.e. might make compiling difficult or impossible, depending on system). High baud-rate CRT terminals are most convenient for use, although any similar ASCII device is useable.

## Appendix 1: Pted source.

A listing of the source program for the editor follows. As mentioned previously, pted is written in PT, a fully compatible subset of Standard Pascal as defined in [Jensen & Wirth74]. An attempt was made to follow strictly a readable paragraphing convention, and comments were used at necessary points, although they were not used as a cheap substitute for self-documenting code. If the code looks a bit Fortran-ish, it is because the original identifiers, structure etc. in [Kernighan & Plauger76] were preserved as much as possible to take advantage of this good existing documentation. Their original code was of course written in RATFOR, a structured Fortran, and some strained identifiers are inevitable.

```
{    PTED is a portable UNIX-style text editor written entirely in PT,        }
{    a subset of Standard Pascal. It is is based on the editor of Kernighan   }
{    and Plauger given in "Software Tools", Addison-Wesley, 1976.             }

{    Author:  Ken Newman                  }
{    Date completed:  Apr. 18, 1980       }

 program editor(input,output,maininp,scr1,scr2,r1,r2,w1,w2);

     const

         { general constants }

         appendcom = 97;      { 'a' }
         backscan = 63;     { '?' }
         backward = 222;      { flag to indicate backward search }
         blank = 32;     { ' ' }
         bufent = 5;     { size of pointer block in buf }
         change = 99;      { 'c' }
         comma = 44;     { ',' }
         curline = 46;      { '.' }
         delcom = 100;      { 'd' }
         eoff = -999;      { end-of-file flag }
         eos = -888;     { end-of-string; ends all strings }
         err = -666;     { error status flag }
         exclude = 118;      { 'v' }
         forwrd = 333;      { flag to indicate forward search }
         global = 103;      { 'g' }
         insert = 105;      { 'i' }
         lastline = 36;      { '$' }
         leng = 4;     { fifth buf entry }
         letn = 110;      { 'n' }
         lett = 116;      { 't' }
         line0 = 1;     { offset of null line 0 }
         mark = 2;     { third entry in buf - used by globals }
         maxbuf = 8000;      { limit of pointer buffer size }
         maxline = 120;      { max. line length }
         maxoffset = 32000;      { offset to allow large file char. offsets }
         maxpat = 175;     { max. size of encoded pattern }
         maxtext = 2000;      { limit to # chars buffered in memory }
```

```
movecom = 109;     { 'm' }
minus = 45;     { '-' }
newline = 10;     { serves as CR-LF on Unix }
next = 1;     { second entry in buf - points to next line }
no = -334;     { flag, opposite of yes }
nostatus = 321;     { flag indicating status undetermined yet }
nott = 94;     { '^' }
ok = -777;     { status flag indicating success }
pageprint = 38;     { '&' }
pagesize = 18;     { # lines printed by '&' }
period = 46;     { '.' }
plus = 43;     { '+' }
prev = 0;     { first entry in buf - points to previous line }
printcur = 61;     { '=' }
print = 112;     { 'p' }
promptchar = 42;     { '*' }
quit = 113;     { 'q' }
readcom = 114;     { 'r' }
scan = 47;     { '/' }
seekadr = 3;     { fourth entry in buf - offset of line on scratch files }
semicol = 59;     { ';' }
substitute = 115;     { 's' }
tab = 9;     { horizontal tab }
writecom = 119;     { 'w' }
yes = -333;     { flag, opposite of no }

{ pattern matching constants }

andd = 38;     { '&' }
any = 46;     { '.' }
bol = 94;     { '^' }
ccl = 91;     { '[' }
cclend = 93;     { ']' }
ch = -10;     { signals presence of char, in encoded patterns }
closize = 4;     { size of closure entry in encoded pattern }
closure = 42;     { '*' }
count = 1;     { rel. offset in encoded pattern for counts }
dash = 45;     { '-' }
ditto = -667;     { indicates '&' char in substitution pattern }
eol = 36;     { 's' }
```

```
    escape = 92;     { '\' }
    nccl = 110;     { indicates negated char. class in encoded pattern }
    prevcl = 2;     { offset of last closure in encoded pattern }
    start = 3;     { offset where a match starts in input line }

type
    alphabet = array [1..27] of integer;     { for holding ASCII alphabet }
    buffer = array [1..maxbuf] of integer;     { for pointer block array }
    digits = array [1..11] of integer;     { for holding ASCII digits }
    lyne = array [1..maxline] of integer;     { input line type }
    pattern = array [1..maxpat] of integer;     { encoded pattern type }
    textf = file of char;     { type of all ext. files }

var
    break: boolean;     { flag for implementing true break stmt }
    buf: buffer;     { holds pointer blocks for lines }
    curln: integer;     { current line }
    cursav: integer;     { saved value of curln }
    dijits: digits;     { holds ASCII digits }
    errmsgchar: char;     { single char. printed with '?' in error msgs }
    etx: integer;     { pointer to last char. buffered in 'text' array }
    fastread: boolean;     { flag to speed up reading on initial start-up only }
    funcval1: integer;     { temp. function return value }
    funcval2: integer;     { temp. function return value }
    funcval3: integer;     { temp. function return value }
    i: integer;     { loop counter in main routine }
    inpempty: integer;     { flag to indicate empty read file }
    lastbf: integer;     { offset of last pointer in buf }
    lastln: integer;     { last line }
    lin: lyne;     { holds input line }
    line1: integer;     { first line number in line number list }
    line2: integer;     { second line number in line number list }
    lowalf: alphabet;     { holds lower-case ASCII alphabet }
    maininp: textf;     { main input file }
    nlines: integer;     { number of line numbers in line number list }
    omatchmsg: lyne;     { for "impossible" error msg in omatch }
    pat: pattern;     { holds encoded pattern }
    patsizmsg: lyne;     { for "impossible" error msg in patsiz }
    prompt: lyne;     { holds prompt }
    r1: textf;     { first auxiliary read file }
```

```
        r2: textf;    { second auxiliary read file }
        realscrend: integer;    { offset of physical end of current scratch file }
        scr1: textf;    { first scratch file }
        scr2: textf;    { second scratch file }
        scrend: integer;    { offset of logical end of current scratch file }
        scrptr: integer;    { pointer into current physical scratch file }
        status: integer;    { holds command return status }
        text: array [1..maxtext] of integer;    { holds buffered text }
        txt: lyne;    { holds single line of text retrieved from scratch file }
        upalf: alphabet;    { holds upper-case ASCII alphabet }
        w1: textf;    { first auxiliary write file }
        w2: textf;    { second auxiliary write file }
        whichscr: integer;    { tells which scratch file (1 or 2) is currently in use }

        { procedures follow }

    { alfindex - find character c in string str }
procedure alfindex(var str: alphabet; c: integer; var retval: integer);
    var i: integer;
    begin
        i := 1;
        while (str[i] <> eos) and (str[i] <> c) do
            i := i+1;
        if (str[i] = eos)
            then retval := 0
            else retval := i
    end;  { alfindex }

    { digindex - find character c in string str }
procedure digindex(var str: digits; c: integer; var retval: integer);
    var i: integer;
    begin
        i := 1;
        while (str[i] <> eos) and (str[i] <> c) do
            i := i+1;
        if (str[i] = eos)
            then retval := 0
            else retval := i
    end;  { digindex }
```

```
   { lynaddset - put c in sett[j] if it fits, increment j }
procedure lynaddset(c: integer; var sett: lyne; var j: integer;
          maxsiz: integer; var retval: integer);
    begin
        if (j > maxsiz)
            then retval := no
            else begin
                     sett[j] := c;
                     j := j+1;
                     retval := yes
                 end
    end;  { lynaddset }

   { pataddset - put c in sett[j] if it fits, increment j }
procedure pataddset(c: integer; var sett: pattern; var j: integer;
          maxsiz: integer; var retval: integer);
    begin
        if (j > maxsiz)
            then retval := no
            else begin
                     sett[j] := c;
                     j := j+1;
                     retval := yes
                 end
    end;  { pataddset }

   { length - compute length of string }
procedure length(var str: lyne; var retval: integer);
    begin
        retval := 0;
        while (str[retval+1] <> eos) do
            retval := retval+1
    end;  { length }

   { readf - read count characters into global txt either from }
   {         memory or a scratch file, depending on offset     }
procedure readf(count: integer; offset: integer);
    var  c: char;  i: integer;  j: integer;
    begin
        i := 1;
```

```
if (offset >= realscrend)
    then begin
             j := offset-realscrend;
             while (count > 0) do
                 begin
                     txt[i] := text[i+j];
                     count := count-1;
                     i := i+1
                 end
         end
    else case whichscr of
             1: while (count > 0) and (not eof(scr1)) do
                    begin
                        if (eoln(scr1))
                            then begin
                                     readln(scr1);
                                     txt[i] := newline
                                 end
                            else begin
                                     read(scr1,c);
                                     txt[i] := ord(c)
                                 end;
                        count := count-1;
                        i := i+1;
                        scrptr := scrptr+1
                    end;
             2: while (count > 0) and (not eof(scr2)) do
                    begin
                        if (eoln(scr2))
                            then begin
                                     readln(scr2);
                                     txt[i] := newline
                                 end
                            else begin
                                     read(scr2,c);
                                     txt[i] := ord(c)
                                 end;
                        count := count-1;
                        i := i+1;
                        scrptr := scrptr+1
```

```
                                end
                        end  { case }
    end;  { readf }

  { getind - locate line index in buffer }
procedure getind(line: integer; var retval: integer);
    var j: integer;   k: integer;
    begin
        k := line0;
        j := 0;
        while (j < line) do
            begin
                k := buf[k+next];
                j := j+1
            end;
        retval := k
    end;  { getind }

    { seek - position current scratch file at offset in preparation     }
    { for reading. If forward seek, read forward from current           }
    { position; if backward, reset to beginning and read forward        }
    { from there.                                                       }
procedure seek(offset: integer);
    var  c: char;
    begin
        if (offset < realscrend)
            then begin
                    if (offset < scrptr)
                        then begin
                                if (whichscr = 1)
                                    then reset(scr1)
                                    else reset(scr2);
                                scrptr := -maxoffset
                             end;
                    case whichscr of
                        1: while (offset > scrptr) and (not eof(scr1)) do
                               begin
                                   if (eoln(scr1))
                                       then readln(scr1)
                                       else read(scr1,c);
```

```
                                scrptr := scrptr+1
                            end;
                        2: while (offset > scrptr) and (not eof(scr2)) do
                            begin
                                if (eoln(scr2))
                                    then readln(scr2)
                                    else read(scr2,c);
                                scrptr := scrptr+1
                            end
                    end  { case }
                end
    end;  { seek }

   { gettxt - locate text for line, copy to txt }
procedure gettxt(line: integer; var retval: integer);
    var  j: integer;  k: integer;
    begin
        getind(line,k);
        seek(buf[k+seekadr]);
        readf(buf[k+leng],buf[k+seekadr]);
        j := buf[k+leng]+1;
        txt[j] := eos;
        retval := k
    end;  { gettxt }

   { relink - rewrite two half links }
procedure relink(a: integer; x: integer; y: integer; b: integer);
    begin
        buf[x+prev] := a;
        buf[y+next] := b
    end;  { relink }

   { getline - read a line from terminal, terminate with newline }
   { and eos                                                      }
procedure getline(var line: lyne; var retval: integer);
    var c: char; i: integer;
    begin
        i := 1;
        while (not eoln) and (i <= maxline-2) do
            begin
```

```
                read(c);
                line[i] := ord(c);
                i := i+1
            end;
        line[i] := newline;
        line[i+1] := eos;
        retval := i;
        readln
    end;  { getline }

  { getlin - read a line from fyle, terminate with newline and eos }
procedure getlin(var line: lyne; var fyle: textf; var retval: integer);
    var  c: char;  i: integer;
    begin
        i := 1;
        if (eof(fyle))
            then retval := eoff
            else begin
                    while (not eoln(fyle))
                          and (not eof(fyle))
                          and (i <= maxline-2) do
                        begin
                            read(fyle,c);
                            line[i] := ord(c);
                            i := i+1
                        end;
                    line[i] := newline;
                    line[i+1] := eos;
                    if (not eoln(fyle))
                        then retval := eoff
                        else retval := i;
                    if (not eof(fyle))
                        then readln(fyle)
                end
    end;  { getlin }

  { remark - print 2 chars on terminal                    }
procedure remark(remark1: char; remark2: char);
    begin
        write(remark1, remark2, chr(newline))
```

```
    end;  { remark }

   { putline - print a line on terminal                          }
procedure putline(var line: lyne);
    var i: integer;
    begin
        i := 1;
        while (line[i] <> eos) and (i <= maxline-1) do
            begin
                write(chr(line[i]));
                i := i+1
            end
    end;  { putline }

   { putlin - write a line onto fyle                             }
procedure putlin(var line: lyne; var fyle: textf);
    var  i: integer;
    begin
        i := 1;
        while (line[i] <> eos) and (i <= maxline-1) do
            begin
                write(fyle,chr(line[i]));
                i := i+1
            end
    end;  { putlin }

   { copyscr - copy directly fromfile onto tofile      }
procedure copyscr(var fromfile: textf; var tofile: textf);
    var  c: char;
    begin
        reset(fromfile);
        rewrite(tofile);
        while (not eof(fromfile)) do
            begin
                while (not eoln(fromfile)) do
                    begin
                        read(fromfile,c);
                        write(tofile,c)
                    end;
                readln(fromfile);
```

```
                writeln(tofile)
            end
    end;  { copyscr }

   { store - store line in global txt into memory if it fits; if }
   { not, copy scratch files and append memory buffer at         }
   { end, then put line into memory. If starting up on           }
   { main file, bypass buffering and write directly on           }
   { scratch file.                                               }
procedure store(linesize: integer);
    var  i: integer;  j: integer;  lin: lyne;
    begin
        if (etx+linesize > maxtext)
            then begin
                    i := 1;
                    whichscr := 3-whichscr;
                    if (whichscr = 1)
                        then copyscr(scr2,scr1)
                        else copyscr(scr1,scr2);
                    while (i <= etx) do
                        begin
                            j := 1;
                        while (text[i] <> newline) do
                            begin
                                lin[j] := text[i];
                                i := i+1;
                                j := j+1
                            end;
                        lin[j] := newline;
                        lin[j+1] := eos;
                        i := i+1;
                        if (whichscr = 1)
                            then putlin(lin,scr1)
                            else putlin(lin,scr2)
    end;
realscrend := scrend;
etx := 0;
if (whichscr = 1)
    then reset(scr1)
    else reset(scr2);
```

```
                    scrptr := -maxoffset
                end;
        if (fastread)
            then case whichscr of
                    1: putlin(txt,scr1);
                    2: putlin(txt,scr2)
                end  { case }
            else begin
                    i := 1;
                    while (txt[i] <> eos) do
                        begin
                            text[etx+i] := txt[i];
                            i := i+1
                        end;
                    etx := etx+linesize
                end
    end;  { store }

{ maklin - make new line entry, copy text to scratch }
{          or to in-core buffer                       }
procedure maklin(var lin: lyne; i: integer; var newind: integer;
                 var retval: integer);
    var  j: integer;  junk: integer;  txtend: integer;  break: boolean;
         funcval: integer;
    begin
        break := false;
        retval := err;
        if (lastbf+bufent <= maxbuf)
            then begin
                    txtend := 1;
                    j := i;
                    while (lin[j] <> eos) and (not break) do
                        begin
                            lynaddset(lin[j],txt,txtend,maxline,junk);
                            j := j+1;
                            if (lin[j-1] = newline)
                                then break := true
                        end;
                    lynaddset(eos,txt,txtend,maxline,funcval);
                    if (funcval <> no)
```

```
                        then begin
                                if (screnд < maxoffset)
                                    then begin
                                            seek(screnд);
                                            buf[lastbf+seekadr] := screnд;
                                            length(txt,funcval);
                                            buf[lastbf+leng] := funcval;
                                            store(buf[lastbf+leng]);
                                            screnд := screnд+buf[lastbf+leng];
                                            buf[lastbf+mark] := no;
                                            newind := lastbf;
                                            lastbf := lastbf+bufent;
                                            retval := j
                                        end
                                    else errmsgchar := 'T'
                                end
                    end
                else errmsgchar := 'T'
        end;  { maklin }

    { setbuf - do some initializing, set up line 0        }
    procedure setbuf;
        var  null: lyne;  k: integer;  junk: integer;
        begin
            screnд := -maxoffset;
            etx := 0;
            whichscr := 2;
            scrptr := -maxoffset;
            realscrend := -maxoffset;
            null[1] := eos;
            lastbf := line0;
            maklin(null,1,k,junk);
            relink(k,k,k,k);
            curln := 0;
            lastln := 0
        end;  { setbuf }

    { prevln - get line before "line"                     }
    procedure prevln(line: integer; var retval: integer);
        begin
```

```
        if (line-1 < 0)
            then retval := lastln
            else retval := line-1
    end;  { prevln }

   { nextln - get line after "line"                          }
procedure nextln(line: integer; var retval: integer);
    begin
        if (line+1 > lastln)
            then retval := 0
            else retval := line+1
    end;  { nextln }

   { inject - insert lin after curln, write scratch    }
   {          or in-core buffer                        }
procedure inject(var lin: lyne; var retval: integer);
    var  i: integer;  k1: integer;  k2: integer;  k3: integer;
         break: boolean;  funcval: integer;
    begin
        i := 1;
        break := false;
        while (lin[i] <> eos) and (not break) do
            begin
                maklin(lin,i,k3,i);
                if (i = err)
                    then begin
                            retval := err;
                            i := 1;
                            break := true
                         end
                    else begin
                            getind(curln,k1);
                            nextln(curln,funcval);
                            getind(funcval,k2);
                            relink(k1,k3,k3,k2);
                            relink(k3,k2,k1,k3);
                            curln := curln+1;
                            lastln := lastln+1;
                            retval := ok
                         end
```

```
            end
    end;  { inject }

   { clrbuf - (for completeness) do some cleanup        }
procedure clrbuf;
    begin
    end;  { clrbuf }

   { esc - map arr[i] into escaped character if appropriate }
procedure esc(var arr: lyne; var i: integer; var retval: integer);
    begin
        if (arr[i] <> escape)
            then retval := arr[i]
            else if (arr[i+1] = eos)
                    then retval := escape
                    else begin
                            i := i+1;
                            if (arr[i] = letn)
                                then retval := newline
                                else if (arr[i] = lett)
                                    then retval := tab
                                    else retval := arr[i]
                         end
    end;  { esc }

   { patsiz - returns size of pattern entry at pat[n]  }
procedure patsiz(var pat: pattern; n: integer; var retval: integer);
    begin
        if (pat[n] = ch)
            then retval := 2
            else if (pat[n] = bol) or (pat[n] = eol) or (pat[n] = any)
                    then retval := 1
                    else if (pat[n] = ccl) or (pat[n] = nccl)
                            then retval := pat[n+1]+2
                            else if (pat[n] = closure)
                                    then retval := closize
                                    else putline(patsizmsg)
    end;  { patsiz }

   { locate - look for c in char. class at patt[offset]        }
```

```
procedure locate(c: integer; var patt: pattern; offset: integer;
          var retval: integer);
    var i: integer;  found: boolean;
    begin
        i := offset+patt[offset];
        found := false;
        retval := no;
        while (i > offset) and (not found) do
            if (c = patt[i])
                then begin
                        retval := yes;
                        found := true
                     end
                else i := i-1
    end;  { locate }

   { omatch - try to match a single pattern at patt[j] }
procedure omatch(var line: lyne; var i: integer; var patt: pattern;
          j: integer; var retval: integer);
    var bump: integer;  locval: integer;  pattj:integer;
    begin
    retval := no;
    if (line[i] <> eos)
        then begin
                bump := -1;
                pattj := patt[j];
                if (pattj = ch) or (pattj = bol) or
                   (pattj = any) or (pattj = eol) or
                   (pattj = ccl) or (pattj = nccl)
                    then case patt[j] of
                            ch:  if (line[i] = patt[j+1])
                                    then bump := 1;
                            bol:  if (i = 1)
                                     then bump := 0;
                            any:  if (line[i] <> newline)
                                     then bump := 1;
                            eol:  if (line[i] = newline)
                                     then bump := 0;
                            ccl:  begin
                                     locate(line[i],patt,j+1,locval);
```

```
                                        if (locval = yes)
                                            then bump := 1
                                    end;
                                nccl:  begin
                                            locate(line[i],patt,j+1,locval);
                                            if (line[i] <> newline) and
                                            (locval = no)
                                             then bump := 1
                                    end
                            end  { case }
                        else putline(omatchmsg);
                    if (bump >= 0)
                        then begin
                                i := i+bump;
                                retval := yes
                            end
                end
    end;  { omatch }

   { amatch - look for match starting at line[from]     }
procedure amatch(var line: lyne; from: integer; var patt: pattern;
      var retval: integer);
var  i: integer;  j: integer;  offset: integer;  stack: integer;
     break: boolean;  funcval: integer;  return: boolean;
begin
    stack := 0;
    offset := from;
    return := false;
    j := 1;
    while (patt[j] <> eos) and (not return) do
        begin
            if (patt[j] = closure)
                then begin
                        stack := j;
                        break := false;
                        j := j+closize;
                        i := offset;
                        while (line[i] <> eos) and (not break) do
                            begin
                                omatch(line,i,patt,j,funcval);
```

```
                    if (funcval = no)
                        then break := true
                end;
            patt[stack+count] := i-offset;
            patt[stack+start] := offset;
            offset := i
        end
    else begin
            omatch(line,offset,patt,j,funcval);
            if (funcval = no)
                then begin
                        break := false;
                        while (stack > 0) and (not break) do
                            if (patt[stack+count] > 0)
                                then break := true
                                else stack := patt[stack+
                                                   prevcl];
                        if (stack <= 0)
                            then begin
                                    retval := 0;
                                    return := true
                                 end
                            else begin
                                    patt[stack+count] :=
                                      patt[stack+count]-1;
                                    j := stack+closize;
                                    offset := patt[stack+
                                      start]+patt[stack+
                                        count]
                                 end
                     end
                end;
            if (not return)
                then begin
                        patsiz(patt,j,funcval);
                        j := j+funcval
                     end
        end;
    if (not return)
        then retval := offset
```

```
    end;  { amatch }

   { match - find match anywhere on line }
procedure match(var line: lyne; var patt: pattern; var retval: integer);
    var i: integer;  found: boolean;  amatchval: integer;
    begin
        i := 1;
        found := false;
        retval := no;
        while (line[i] <> eos) and (not found) do
            begin
                amatch(line,i,patt,amatchval);
                if (amatchval > 0)
                    then begin
                                retval := yes;
                                found := true
                         end
                    else i := i+1
            end
    end;  { match }

   { ptscan - scan for next occurrence of pattern      }
procedure ptscan(way: integer; var num: integer; var retval: integer);
    var k: integer;  found: boolean;  matchval: integer;
    begin
        num := curln;
        found := false;
        retval := err;
        repeat
            if (way = forwrd)
                then nextln(num,num)
                else prevln(num,num);
            gettxt(num,k);
            match(txt,pat,matchval);
            if (matchval = yes)
                then begin
                            retval := ok;
                            found := true
                     end
        until found or (num = curln);
```

```
        if (not found) and (errmsgchar = '.')
           then errmsgchar := '/'
    end;  { ptscan }

   { alfdodash - expand arr[i-1]-arr[i+1] into sett[j]...  }
   {             from valid                                }
procedure alfdodash(var valid: alphabet; var arr: lyne;
      var i: integer; var sett: pattern; var j: integer; maxset: integer);
    var junk: integer;  k: integer;  limit: integer;  funcval: integer;
    begin
        i := i+1;
        j := j-1;
        esc(arr,i,funcval);
        alfindex(valid,funcval,limit);
        alfindex(valid,sett[j],k);
        while (k <= limit) do
            begin
                pataddset(valid[k],sett,j,maxset,junk);
                k := k+1
            end
    end;  { alfdodash }

   { digdodash - expand arr[i-1]-arr[i+1] into sett[j]...   }
   {             from valid                                 }
procedure digdodash(var valid: digits; var arr: lyne; var i: integer;
          var sett: pattern; var j: integer; maxset: integer);
    var junk: integer;  k: integer;  limit: integer;  funcval: integer;
    begin
        i := i+1;
        j := j-1;
        esc(arr,i,funcval);
        digindex(valid,funcval,limit);
        digindex(valid,sett[j],k);
        while (k <= limit) do
            begin
                pataddset(valid[k],sett,j,maxset,junk);
                k := k+1
            end
    end;  { digdodash }
```

```
    { filset - expand set at arr[i] into sett[j], stop at delim }
procedure filset(delim: integer; var arr: lyne; var i: integer;
          var sett: pattern; var j: integer; maxset: integer);
    var junk: integer;  funcval: integer;
    begin
        while (arr[i] <> delim) and (arr[i] <> eos) do
            begin
                if (arr[i] = escape)
                    then begin
                            esc(arr,i,funcval);
                            pataddset(funcval,sett,j,maxset,junk)
                         end
                else if (arr[i] <> dash)
                            then pataddset(arr[i],sett,j,maxset,junk)
                else if (j <= 1) or (arr[i+1] = delim)
                            then pataddset(dash,sett,j,maxset,junk)
                else begin
                            digindex(dijits,sett[j-1],funcval);
                            if (funcval > 0)
                                then digdodash(dijits,arr,i,sett,j,maxset)
                            else begin
                                    alfindex(lowalf,sett[j-1],funcval);
                                    if (funcval > 0)
                                        then alfdodash(lowalf,arr,i,sett,
                                                       j,maxset)
                                        else begin
                                            alfindex(upalf,sett[j-1],funcval);
                                            if (funcval > 0)
                                                then alfdodash(upalf,arr,i,sett,
                                                               j,maxset)
                                                else pataddset(dash,sett,j,
                                                               maxset,junk)
                                             end
                                 end
                     end;
                i := i+1
            end
    end;  { filset }

{ getccl - expand char. class at arg[i] into patt[j]          }
```

```
procedure getccl(var arg: lyne; var i: integer; var patt: pattern;
          var j: integer;  var retval: integer);
    var jstart: integer;  junk: integer;
    begin
        i := i+1;
        if (arg[i] = nott)
            then begin
                    pataddset(nccl,patt,j,maxpat,junk);
                    i := i+1
                end
            else pataddset(ccl,patt,j,maxpat,junk);
        jstart := j;
        pataddset(0,patt,j,maxpat,junk);
        filset(cclend,arg,i,patt,j,maxpat);
        patt[jstart] := j-jstart-1;
        if (arg[i] = cclend)
            then retval := ok
            else retval := err
    end;  { getccl }

  { stclos - insert closure entry at patt[j]          }
procedure stclos(var patt: pattern; var j: integer; var lastj: integer;
          lastcl: integer; var retval: integer);
    var jp: integer;  jt: integer;  junk: integer;
    begin
        jp := j-1;
        while (jp >= lastj) do
            begin
                jt := jp+closize;
                pataddset(patt[jp],patt,jt,maxpat,junk);
                jp := jp-1
            end;
        j := j+closize;
        retval := lastj;
        pataddset(closure,patt,lastj,maxpat,junk);
        pataddset(0,patt,lastj,maxpat,junk);
        pataddset(lastcl,patt,lastj,maxpat,junk);
        pataddset(0,patt,lastj,maxpat,junk)
    end;  { stclos }
```

```
    { makpat - make pattern from arg[from], terminate at delim }
procedure makpat(var arg: lyne; from: integer; delim: integer;
          var patt: pattern; var retval: integer);
    var i: integer;  j: integer;  junk: integer;  lastcl: integer;
        lastj: integer;  lj: integer;  break: boolean;  funcval: integer;
    begin
        j := 1;
        lastj := 1;
        lastcl := 0;
        break := false;
        i := from;
        while (arg[i] <> delim) and (arg[i] <> eos) and (not break) do
            begin
                lj := j;
                if (arg[i] = any)
                    then pataddset(any,patt,j,maxpat,junk)
                else if (arg[i] = bol) and (i = from)
                        then pataddset(bol,patt,j,maxpat,junk)
                else if (arg[i] = eol) and (arg[i+1] = delim)
                        then pataddset(eol,patt,j,maxpat,junk)
                else if (arg[i] = ccl)
                    then begin
                            getccl(arg,i,patt,j,funcval);
                            if (funcval = err)
                                then break := true
                        end
            else if (arg[i] = closure) and (i > from)
                    then begin
                            lj := lastj;
                            if (lj >= maxpat)
                                then break := true
                                else begin
                                        if (patt[lj] = bol) or (patt[lj] = eol) or
                                           (patt[lj] = closure)
                                            then break := true
                                            else stclos(patt,j,lastj,lastcl,lastcl)
                                    end
                        end
            else begin
                    pataddset(ch,patt,j,maxpat,junk);
```

```
                        esc(arg,i,funcval);
                        pataddset(funcval,patt,j,maxpat,junk)
                    end;
                if (not break)
                    then begin
                            lastj := lj;
                            i := i+1
                        end
            end;
        if (arg[i] <> delim)
            then retval := err
            else begin
                    pataddset(eos,patt,j,maxpat,funcval);
                    if (funcval = no)
                        then retval := err
                        else retval := i
                end
    end;  { makpat }

   { optpat - make pattern if specified at line[i]      }
procedure optpat(var line: lyne; var i: integer; var retval: integer);
begin
    if (line[i] = eos)
        then i := err
        else if (line[i+1] = eos)
                then i := err
                else if (line[i+1] = line[i])
                        then i := i+1
                        else makpat(line,i+1,line[i],pat,i);
    if (pat[1] = eos)
        then i := err;
    if (i = err)
        then begin
                pat[1] := eos;
                if (errmsgchar = '.')
                    then errmsgchar := 'p';
                retval := err
            end
        else retval := ok
end;  { optpat }
```

```
   { maksub - make substitution string in sub          }
procedure maksub(var arg: lyne; from: integer; delim: integer;
          var sub: pattern; var retval: integer);
    var i: integer;  j: integer;  junk: integer;  funcval: integer;
    begin
        j := 1;
        i := from;
        while (arg[i] <> delim) and (arg[i] <> eos) do
            begin
                if (arg[i] = andd)
                    then pataddset(ditto,sub,j,maxpat,junk)
                    else begin
                             esc(arg,i,funcval);
                             pataddset(funcval,sub,j,maxpat,junk)
                         end;
                i := i+1
            end;
        if (arg[i] <> delim)
            then retval := err
            else begin
                     pataddset(eos,sub,j,maxpat,funcval);
                     if (funcval = no)
                         then retval := err
                         else retval := i
                 end
    end;  { maksub }

   { catsub - add replacement text to end of new       }
procedure catsub(var line: lyne; from: integer; too: integer;
          var sub: pattern; var new: lyne; var k: integer;
          maxnew: integer);
    var i: integer;  j: integer;  junk: integer;
    begin
        i := 1;
        while (sub[i] <> eos) do
            begin
                if (sub[i] = ditto)
                    then begin
                             j := from;
```

```
                        while (j < too) do
                            begin
                                lynaddset(line[j],new,k,maxnew,junk);
                                j := j+1
                            end
                    end
                else lynaddset(sub[i],new,k,maxnew,junk);
            i := i+1
        end
end;  { catsub }

  { getrhs - get substitution string for "s" command  }
procedure getrhs(var line: lyne; var i: integer; var sub: pattern;
        var gflag: integer; var retval: integer);
begin
    retval := err;
    if (line[i] <> eos)
        then if (line[i+1] <> eos)
            then begin
                    maksub(line,i+1,line[i],sub,i);
                    if (i <> err)
                then begin
                        if (line[i+1] = global)
                            then begin
                                    i := i+1;
                                    gflag := yes
                                end
                            else gflag := no;
                        retval := ok
                    end
                end
end;  { getrhs }

  { itoc - convert integer int to char. string in str          }
procedure itoc(int: integer; var str: lyne; size: integer;
        var retval: integer);
    var d: integer;  i:integer;  intval: integer;  j:integer;
        k: integer;
    begin
        if (int < 0)
```

```
        then intval := -int
        else intval := int;
    str[1] := eos;
    i := 1;
    repeat
        i := i+1;
        d := intval mod 10;
        str[i] := dijits[d+1];
        intval := intval div 10
    until (intval = 0) or (i >= size);
    if (int < 0) and (i < size)
        then begin
                 i := i+1;
                 str[i] := minus
             end;
    retval := i-1;
    j := 1;
    while (j < i) do
        begin
            k := str[i];
            str[i] := str[j];
            str[j] := k;
            i := i-1;
            j := j+1
        end
    end;  { itoc }

  { skipbl - skip blanks and tabs at line[i]...        }
procedure skipbl(var line: lyne; var i: integer);
    begin
        while (line[i] = blank) or (line[i] = tab) do
            i := i+1
    end;  { skipbl }

  { ctoi - convert string at inn[i] to integer, increment i }
procedure ctoi(var inn: lyne; var i: integer; var retval: integer);
    var  d: integer;  break: boolean;
    begin
        break := false;
        while (inn[i] = blank) or (inn[i] = tab) do
```

```
            i := i+1;
        retval := 0;
        while (inn[i] <> eos) and (not break) do
            begin
                digindex(dijits,inn[i],d);
                if (d = 0)
                    then break := true
                    else begin
                            retval := 10*retval+d-1;
                            i := i+1
                        end
            end
    end;  { ctoi }

   { getnum - convert one term to line number          }
procedure getnum(var line: lyne; var i: integer; var pnum: integer;
          var status: integer; var retval: integer);
    var  funcval: integer;
    begin
        retval := ok;
        digindex(dijits,line[i],funcval);
if (funcval > 0)
    then begin
            ctoi(line,i,pnum);
            i := i-1
        end
    else if (line[i] = plus) or (line[i] = minus)
            then begin
                    pnum := curln;
                    i := i-1
                end
    else if (line[i] = curline)
            then pnum := curln
    else if (line[i] = lastline)
            then pnum := lastln
    else if (line[i] = scan) or (line[i] = backscan)
            then begin
                    optpat(line,i,funcval);
                    if (funcval = err)
                        then retval := err
```

```
                                    else if (line[i] = scan)
                                            then ptscan(forwrd,pnum,retval)
                                            else ptscan(backward,pnum,retval)
                                end
                else retval := eoff;
            if (retval = ok)
                then i := i+1;
            status := retval
        end;  { getnum }

    { getone - evaluate one line number expression      }
procedure getone(var line: lyne; var i: integer; var num: integer;
          var status: integer; var retval: integer);
        var  istart: integer;  mul: integer;  pnum: integer;  break: boolean;
             funcval: integer;
        begin
            istart := i;
            break := false;
            num := 0;
            skipbl(line,i);
            getnum(line,i,num,status,funcval);
            if (funcval = ok)
                then repeat
                        skipbl(line,i);
                        if (line[i] <> plus) and (line[i] <> minus)
                            then begin
                                    status := eoff;
                                    break := true
                                end;
                        if (not break)
                            then begin
                                    if (line[i] = plus)
                                        then mul := 1
                                        else mul := -1;
                                    i := i+1;
                                    skipbl(line,i);
                                    if (line[i] = plus) or (line[i] = minus)
                                        then status := err
                                        else begin
                                                getnum(line,i,pnum,status,funcval);
```

```
                                                    if (funcval = ok)
                                                        then num := num+mul*pnum;
                                                    if (status = eoff)
                                                        then status := err
                                                end
                            end
                    until break or (status <> ok);
        if (num < 0) or (num > lastln)
            then begin
                    status := err;
                    if (errmsgchar = '.')
                        then errmsgchar := '$'
                end;
        if (status = err)
            then retval := err
            else if (i <= istart)
                    then retval := eoff
                    else retval := ok;
        status := retval
    end;  { getone }

    { getlst - collect line numbers (if any) at line[i],          }
    {          increment i                                        }
procedure getlst(var line: lyne; var i: integer; var status: integer;
          var retval: integer);
    var num: integer;  break: boolean;  funcval: integer;
    begin
        skipbl(line,i);
        line2 := 0;
        break := false;
        nlines := 0;
        getone(line,i,num,status,funcval);
        while (funcval = ok) and (not break) do
            begin
                line1 := line2;
                line2 := num;
                nlines := nlines+1;
                if (line[i] <> comma) and (line[i] <> semicol)
                    then break := true
                    else begin
```

```
                                if (line[i] = semicol)
                                    then curln := num;
                                i := i+1;
                                getone(line,i,num,status,funcval)
                            end
                end;
            if (nlines > 2)
                then nlines := 2;
            if (nlines = 0)
                then line2 := curln;
            if (nlines <= 1)
                then line1 := line2;
            if (status <> err)
                then status := ok;
            retval := status
        end;  { getlst }

    { defalt - set defaulted line numbers                    }
procedure defalt(def1: integer; def2: integer; var status: integer;
          var retval: integer);
        begin
            if (nlines = 0)
                then begin
                        line1 := def1;
                        line2 := def2
                    end;
            if (line1 > line2) or (line1 <= 0)
                then status := err
                else status := ok;
            retval := status
        end;  { defalt }

    { ckp - check for "p" after command                      }
procedure ckp(var line: lyne; i: integer; var pflag: integer;
          var status: integer; var retval: integer);
        var  j: integer;
        begin
            j := i;
            if (line[j] = print)
                then begin
```

```
                    j := j+1;
                    pflag := yes
                end
            else pflag := no;
        if (line[j] = newline)
            then status := ok
            else status := err;
        retval := status
    end;  { ckp }

  { append - append lines after "line"                    }
procedure append(line: integer; glob: integer; var retval: integer);
    var  lin: lyne;  funcval: integer;
    begin
        if (glob = yes)
            then begin
                    retval := err;
                    if (errmsgchar = '.')
                        then errmsgchar := 'g'
                end
            else begin
                    curln := line;
                    retval := nostatus;
                    while (retval = nostatus) do
                        begin
                            getline(lin,funcval);
                            if (funcval = eoff)
                                then retval := eoff
                                else if (lin[1] = period) and
                                        (lin[2] = newline)
                                         then retval := ok
                                         else begin
                                                 inject(lin,funcval);
                                                 if (funcval = err)
                                                     then retval := err
                                             end
                        end
                end
    end;  { append }
```

```
   { delete - delete lines from through too                    }
procedure delete(from: integer; too: integer; var status: integer;
          var retval: integer);
    var  k1: integer;  k2: integer;  funcval: integer;
    begin
        if (from <= 0)
            then status := err
            else begin
                    prevln(from,funcval);
                    getind(funcval,k1);
                    nextln(too,funcval);
                    getind(funcval,k2);
                    lastln := lastln-(too-from+1);
                    prevln(from,curln);
                    relink(k1,k2,k1,k2);
                    status := ok
                end;
        retval := status
    end;  { delete }

   { move - move line1 through line2 after line3               }
procedure move(var line3: integer; var retval: integer);
    var  k0: integer;  k1: integer;  k2: integer;  k3: integer;
         k4: integer;  k5: integer;  funcval: integer;
    begin
        if (line1 <= 0) or ((line1 <= line3) and (line3 <= line2))
            then begin
                    retval := err;
                    if (errmsgchar = ',')
                        then errmsgchar := 'm'
                end
            else begin
                    prevln(line1,funcval);
                    getind(funcval,k0);
                    nextln(line2,funcval);
                    getind(funcval,k3);
                    getind(line1,k1);
                    getind(line2,k2);
                    relink(k0,k3,k0,k3);
                    if (line3 > line1)
```

```
                    then begin
                            curln := line3;
                            line3 := line3-(line2-line1+1)
                        end
                    else curln := line3+(line2-line1+1);
                getind(line3,k4);
                nextln(line3,funcval);
                getind(funcval,k5);
                relink(k4,k1,k2,k5);
                relink(k2,k5,k4,k1);
                retval := ok
            end
    end;  { move }

   { subst - substitute "sub" for occurrences of pattern }
procedure subst(var sub: pattern; gflag: integer; var retval: integer);
    var  new: lyne;  j: integer;  junk: integer;  k: integer;  lastm: integer;
         line: integer;  m: integer;  status: integer;  subbed: integer;
         break: boolean;  funcval: integer;
    begin
        break := false;
retval := err;
if (line1 > 0)
    then begin
            if (errmsgchar = '.')
                then errmsgchar := 's';
            line := line1;
            while (line <= line2) and (not break) do
                begin
                    j := 1;
                    subbed := no;
                    gettxt(line,junk);
                    lastm := 0;
                    k := 1;
                    while (txt[k] <> eos) do
                        begin
                            if (gflag = yes) or (subbed = no)
                                then amatch(txt,k,pat,m)
                                else m := 0;
                            if (m > 0) and (lastm <> m)
```

```
                then begin
                        subbed := yes;
                        catsub(txt,k,m,sub,new,
                                j,maxline);
                        lastm := m
                    end;
            if (m = 0) or (m = k)
                then begin
                        lynaddset(txt[k],new,j,
                                maxline,junk);
                        k := k+1
                    end
                else k := m
        end;
    if (subbed = yes)
        then begin
                if (errmsgchar = 's')
                    then errmsgchar := '.';
                lynaddset(eos,new,j,maxline,
                                funcval);
                if (funcval = no)
                    then begin
                            retval := err;
                            break := true
                        end;
                if (not break)
                    then begin
                            delete(line,line,
                                status,junk);
                            inject(new,retval);
                            if (retval = err)
                                then break := true
                                else retval := ok
                        end
                end;
            if (not break)
                then line := line+1
        end
    end
end;  { subst }
```

```
   { doprnt - print lines from through too                    }
procedure doprnt(from: integer; too: integer; var retval: integer);
    var  i: integer;  j: integer;
    begin
        if (from <= 0)
            then begin
                    retval := err;
                    if (errmsgchar = '.')
                        then errmsgchar := '$'
                end
            else begin
                    i := from;
                    while (i <= too) do
                        begin
                            gettxt(i,j);
                            putline(txt);
                            i := i+1
                        end;
                    curln := too;
                    retval := ok
                end
    end;  { doprnt }

   { getfn - get file specifier from lin[i]...                 }
procedure getfn(var lin: lyne; i: integer; var filenum: integer;
            var retval: integer);
    var  j: integer;  one: integer;  two: integer;
    begin
        retval := err;
        one := ord('1');
        two := ord('2');
        if (lin[i+1] = blank)
            then begin
                    j := i+2;
                    skipbl(lin,j);
                    if (lin[j] = one) or (lin[j] = two)
                        then begin
                                filenum := lin[j]-one+1;
                                retval := ok
```

```
                        end
                    else if (errmsgchar = '.')
                            then errmsgchar := 'f'
                end
            else if (lin[i+1] = newline)
                    then begin
                            filenum := 0;
                            retval := ok
                         end
                    else if (errmsgchar = '.')
                            then errmsgchar := 'f'
    end;  { getfn }

   { putdec - print number as char. string                    }
procedure putdec(number: integer);
    var  numstring: lyne;  funcval: integer;
    begin
        itoc(number,numstring,maxline,funcval);
        numstring[funcval+1] := newline;
        numstring[funcval+2] := eos;
        putline(numstring)
    end;  { putdec }

   { doread - read "fyle" after "line"                        }
procedure doread(line: integer; var fyle: textf; var inpempty: integer;
            var retval: integer);
    var  lin: lyne;  count: integer;  break: boolean;  funcval: integer;
    begin
        break := false;
        reset(fyle);
        retval := ok;
        if (eof(fyle))
            then begin
                    inpempty := yes;
                    if (errmsgchar = '.')
                        then errmsgchar := 'o'
                end
            else begin
                    inpempty := no;
                    curln := line;
```

```
                    count := 0;
                    getlin(lin,fyle,funcval);
                    while (funcval <> eoff) and (not break) do
                        begin
                            inject(lin,retval);
                            if (retval = err)
                                then break := true
                                else begin
                                        count := count+1;
                                        getlin(lin,fyle,funcval)
                                    end
                        end;
                    putdec(count)
                end
    end;  { doread }

   { dowrit - write "from" through "too" into "fyle"   }
procedure dowrit(from: integer; too: integer; var fyle: textf;
            var retval: integer);
    var  k: integer;  line: integer;
    begin
        rewrite(fyle);
        line := from;
        while (line <= too) do
            begin
                gettxt(line,k);
                putlin(txt,fyle);
                line := line+1
            end;
        putdec(too-from+1);
        retval := ok
    end;  { dowrit }

   { docmd - handle all commands except globals        }
procedure docmd(var lin: lyne; var i: integer; glob: integer;
          var status: integer; var retval: integer);
    var  sub: pattern;  gflag: integer;  line3: integer;  pflag: integer;
         filenum: integer;  inpempty: integer;
         funcval: integer;
    begin
```

```
pflag := no;
status := err;
if (lin[i] = appendcom)
    then begin
            if (lin[i+1] = newline)
             then append(line2,glob,status)
         end
else if (lin[i] = change)
         then begin
                 if (lin[i+1] = newline)
                  then begin
                                defalt(curln,curln,status,funcval);
                                if (funcval = ok)
                                  then begin
                                    if (glob = no)
                                      then begin
                                              delete(line1,line2,status,
                                                     funcval);
                                              if (funcval = ok)
                                                then begin
                                                     prevln(line1,
                                                        funcval);
                                                     append(
                                                        funcval,
                                                        glob,
                                                        status)
                                                end
                                           end
                                      else begin
                                              status := err;
                                              if (errmsgchar = ',')
                                                  then errmsgchar := 'g'
                                           end
                                    end
                           end
                end
else if (lin[i] = delcom)
         then begin
                 ckp(lin,i+1,pflag,status,funcval);
                 if (funcval = ok)
```

```
                        then begin
                                defalt(curln,curln,status,funcval);
                                if (funcval = ok)
                                    then begin
                                            delete(line1,line2,status,
                                                   funcval);
                                            if (funcval = ok)
                                                then begin
                                                        nextln(curln,
                                                            funcval);
                                                        if (funcval<>
                                                                0)
                                                        then nextln(
                                                                curln,
                                                                curln)
                                                     end
                                         end
                            end
                end
    else if (lin[i] = insert)
                then begin
                        if (lin[i+1] = newline)
                            then begin
                                    prevln(line2,funcval);
                                    append(funcval,glob,status)
                                 end
                end
    else if (lin[i] = printcur)
                then begin
                        ckp(lin,i+1,pflag,status,funcval);
                        if (funcval = ok)
                            then begin
                                    putdec(line2)
                                 end
                end
    else if (lin[i] = movecom)
                then begin
                        i := i+1;
                        getone(lin,i,line3,status,funcval);
                        if (funcval = eoff)
```

```
                    then status := err;
                if (status = ok)
                    then begin
                                ckp(lin,i,pflag,status,funcval);
                                if (funcval = ok)
                                    then begin
                                                defalt(curln,curln,
                                                    status,funcval);
                                                if (funcval = ok)
                                                    then move(line3,
                                                                status)
                                                end
                                end
                    end
        else if (lin[i] = substitute)
                then begin
                        i := i+1;
                        optpat(lin,i,funcval);
                        if (funcval = ok)
                            then begin
                                getrhs(lin,i,sub,gflag,funcval);
                                if (funcval = ok)
                                    then begin
                                                ckp(lin,i+1,pflag,status,
                                                    funcval);
                                                if (funcval = ok)
                                                    then begin
                                                                defalt(curln,
                                                                    curln,
                                                                    status,
                                                                    funcval);
                                                                if (funcval =
                                                                    ok)
                                                                then
                                                                  subst(
                                                                    sub,
                                                                    gflag,
                                                                    status)
                                                                end
                                                end
                                    end
```

```
                        end
                end
else if (lin[i] = readcom)
        then begin
                getfn(lin,i,filenum,funcval);
                if (funcval = ok)
                    then begin
                            case filenum of
                               0: doread(line2,maininp,inpempty,
                                                        status);
                               1: doread(line2,r1,inpempty,status);
                               2: doread(line2,r2,inpempty,status)
                            end;  { case }
                            if (inpempty = yes)
                                then status := err
                        end
                end
else if (lin[i] = writecom)
        then begin
                getfn(lin,i,filenum,funcval);
                if (funcval = ok)
                    then begin
                            defalt(1,lastln,status,funcval);
                            if (funcval = ok)
                                then case filenum of
                                       0:  dowrit(line1,
                                                  line2,maininp,
                                                  status);
                                       1: dowrit(line1,line2,w1,
                                                  status);
                                       2: dowrit(line1,line2,w2,
                                                  status)
                                    end  { case }
                        end
                end
else if (lin[i] = print)
        then begin
                if (lin[i+1] = newline)
                    then begin
                            defalt(curln,curln,status,funcval);
```

```
                                    if (funcval = ok)
                                        then doprnt(line1,line2,status)
                                end
                    end
        else if (lin[i] = pageprint)
                then begin
                        if (lin[i+1] = newline) and (nlines = 0) and
                           (glob = no)
                            then begin
                                    if (curln+pagesize > lastln)
                                        then defalt(curln,lastln,status,funcval)
                                        else defalt(curln,curln+pagesize,status,funcval);
                                    if (funcval = ok)
                                        then doprnt(line1,line2,status)
                                end
                    end
        else if (lin[i] = newline)
                then begin
                        if (nlines = 0) and (glob = no)
                            then begin
                                    nextln(curln,line2);
                                    doprnt(line2,line2,status)
                                end
                            else doprnt(line1,line2,status)
                    end
        else if (lin[i] = quit)
                then begin
                        if (lin[i+1] = newline) and (nlines = 0)
                            then if (glob = no)
                                    then status := eoff
                                    else if (errmsgchar = '.')
                                            then errmsgchar := 'g'
                    end
        else if (errmsgchar = '.')
                then errmsgchar := 'x';
        if (status = ok) and (pflag = yes)
            then doprnt(curln,curln,status);
        retval := status
    end;  { docmd }
```

```
   { ckglob - if global prefix, mark lines to be affected }
procedure ckglob(var lin: lyne; var i: integer; var status: integer;
          var retval: integer);
    var  gflag: integer;  k: integer;  line: integer;
         funcval1: integer;  funcval2: integer;
    begin
        if (lin[i] <> global) and (lin[i] <> exclude)
            then status := eoff
            else begin
                    if (lin[i] = global)
                        then gflag := yes
                        else gflag := no;
                    i := i+1;
                    optpat(lin,i,funcval1);
                    defalt(1,lastln,status,funcval2);
                    if (funcval1 = err) or (funcval2 = err)
                        then status := err
                        else begin
                                i := i+1;
                                line := line1;
                                while (line <= line2) do
                                begin
                                    gettxt(line,k);
                                    match(txt,pat,funcval1);
                                    if (funcval1 = gflag)
                                        then buf[k+mark] := yes
                                        else buf[k+mark] := no;
                                    line := line+1
                                end;
                             nextln(line2,line);
                             while (line <> line1) do
                                begin
                                    getind(line,k);
                                    buf[k+mark] := no;
                                    nextln(line,line)
                                end;
                             status := ok
                        end
            end;
retval := status
```

```
    end;  { ckglob }

   { doglob - do command at lin[i] on all marked lines }
procedure doglob(var lin: lyne; var i: integer; var cursav: integer;
          var status: integer; var retval: integer);
    var  count: integer;  istart: integer;  k: integer;  line: integer;
       ' funcval: integer;
    begin
        status := ok;
        count := 0;
        line := line1;
        istart := i;
        repeat
            getind(line,k);
            if (buf[k+mark] = yes)
                then begin
                        buf[k+mark] := no;
                        curln := line;
                        cursav := curln;
                        i := istart;
                        getlst(lin,i,status,funcval);
                        if (funcval = ok)
                            then begin
                                    docmd(lin,i,yes,status,funcval);
                                    if (funcval = ok)
                                        then count := 0
                                end
                    end
                else begin
                        nextln(line,line);
                        count := count+1
                    end
        until (count > lastln) or (status <> ok);
        retval := status
    end;  { doglob }


    { pted main routine }

begin
```

```
i := 0;
while (i < 10) do
    begin
        dijits[i+1] := ord('0')+i;
        i := i+1
    end;
dijits[11] := eos;
i := 0;
while (i < 26) do
    begin
        lowalf[i+1] := ord('a')+i;
        upalf[i+1] := ord('A')+i;
        i := i+1
    end;
lowalf[27] := eos;
upalf[27] := eos;
patsizmsg[1] := ord('?');
patsizmsg[2] := ord('p');
patsizmsg[3] := ord('s');
patsizmsg[4] := newline;
patsizmsg[5] := eos;
omatchmsg[1] := ord('?');
omatchmsg[2] := ord('o');
omatchmsg[3] := ord('m');
omatchmsg[4] := newline;
omatchmsg[5] := eos;
break := false;
errmsgchar := ',';
fastread := false;
setbuf;
pat[1] := eos;
case whichscr of
    1: rewrite(scr1);
    2: rewrite(scr2)
end;  { case }
fastread := true;
doread(0,maininp,inpempty,funcval1);
fastread := false;
if (inpempty = no)
    then begin
```

```
                case whichscr of
                    1: reset(scr1);
                    2: reset(scr2)
                end;  { case }
                realscrend := screnд
            end;
if (funcval1 = err) or (inpempty = yes)
   then begin
            remark('?',errmsgchar);
            errmsgchar := '.'
        end;
prompt[1] := promptchar;
prompt[2] := eos;
putline(prompt);
getline(lin,funcval1);
while (funcval1 <> eoff) and (not break) do
    begin
        i := 1;
        cursav := curln;
        getlst(lin,i,status,funcval2);
if (funcval2 = ok)
    then begin
            ckglob(lin,i,status,funcval3);
            if (funcval3 = ok)
                then doglob(lin,i,cursav,status,status)
                else if (status <> err)
                        then docmd(lin,i,no,status,status)
                        { else error, do nothing }
        end;
if (status = err)
    then begin
            remark('?',errmsgchar);
            errmsgchar := '.';
            curln := cursav
        end;
if (status = eoff)
    then break := true
    else begin
            putline(prompt);
            getline(lin,funcval1)
```

```
                    end
        end;
    clrbuf
end.  { main routine }
```

## Appendix 2: C shell program used on Unix.

For Unix versions of pted, or to give an idea of the essentials of an intermediate program as discussed earlier for improving the user interface, the C version is listed here.

```
     /* C shell program for the Pted text editor                    */
     /* Author: Ken Newman                                          */
     /* Date Completed: April 18, 1980.                             */

     /* The complete syntax of the pted command is                  */
     /*                                                             */
     /*          pted inputfile [-r r1 [r2] ] [-w w1 [w2] ]         */
     /*                                                             */
     /* See the documentation for more details.                     */

#define pmode 0644
char *pted "/u2/newman/bin/pted.load";
char *scr1 "/tmp/ptedaaXXXXX";
char *scr2 "/tmp/ptedbaXXXXX";
char *r1 "/tmp/ptedcaXXXXX";
char *r2 "/tmp/pteddaXXXXX";
char *w1 "w1.$$";
char *w2 "w2.$$";

rmtmp()
{
    execl("/bin/rm","rm",scr1,scr2,r1,r2,0);
    exit(1);
}    /* end of rmtmp */

main(argc,argv)
int argc;
char *argv[];
{
    /* remove temp. files if pted gets killed */
    if (signal(2,rmtmp) == 1)     /* running in background, rubout ignored */
        signal(2,1);              /* continue to ignore rubout */
    if (signal(1,rmtmp) == 1)     /* running nohup, ignore hangup */
        signal(1,1);
    mktemp(scr1);
    create(scr1);
    mktemp(scr2);
    create(scr2);
    mktemp(r1);
    create(r1);
```

```
    mktemp(r2);
    create(r2);

    if (argv[1][0] == '-' && (argv[1][1] == 'r' || argv[1][1] == 'w') &&
                                                    argv[1][2] == 0)
        printusage();
    else if (argc > 1) {
            makesure(argv[1],2);
            getfiles(argc, argv);
            execl("/bin/rm", "rm", scr1, scr2, r1, r2, 0);
        }
        else
            printusage();
}  /* end of main */

getfiles(argc, argv)
int argc;
char *argv[];
{
    int listptr;

    listptr = 2;
    if (argv[listptr][0] == '-' && argv[listptr][1] == 'r' &&
            argv[listptr][2] == 0) {
        listptr = listptr+1;
        if (argc == listptr)
            printusage();
        makesure(argv[listptr],0);
        if (listptr+1 == argc)
            exekute(argv[1],argv[listptr],r2,w1,w2);
        else {
            listptr = listptr+1;
            if (argv[listptr][0] == '-' && argv[listptr][1] == 'w' &&
                    argv[listptr][2] == 0)
                getwfiles(listptr,argc,argv);
            else {
                makesure(argv[listptr],0);
                listptr = listptr+1;
                getwfiles(listptr,argc,argv);
            }
```

```
        }
    }
    else
        getwfiles(listptr,argc,argv);
}  /* end of getfiles */

create(file)
char *file;
{
    if (creat(file,pmode) == -1) {
        printf("pted: can't create %s\n",file);
        rmtmp();
    }
}  /* end of create */

makesure(file,mode)
char *file;
int mode;
{
    if (open(file,mode) == -1) {
        if (creat(file,pmode) == -1) {
            printf("pted: can't create %s\n",file);
            rmtmp();
        }
    }
}  /* end of makesure */

getwfiles(listptr,argc,argv)
int argc;
char *argv[];
int listptr;
{
    int savearg;

    savearg = listptr;
    if (argv[listptr][0] == '-' && argv[listptr][1] == 'w' &&
                        argv[listptr][2] == 0)  {
        listptr = listptr+1;
        if (argc == listptr)
            printusage();
```

```
    if (listptr+1 == argc)
        switch(savearg) {
            case 2: exekute(argv[1],r1,r2,argv[listptr],w2);
                    return;
            case 4: exekute(argv[1],argv[3],r2,argv[listptr],w2);
                    return;
            case 5: exekute(argv[1],argv[3],argv[4],argv[listptr],w2);
                    return;
        }
    else {
        if (listptr+2 != argc)
            printusage();
        switch (savearg) {
            case 2: exekute(argv[1],r1,r2,argv[listptr],argv[listptr+1]);
                    return;
            case 4: exekute(argv[1],argv[3],r2,argv[listptr],argv[listptr+1]);
                    return;
            case 5: exekute(argv[1],argv[3],argv[4],argv[listptr],argv[listptr+1]);
                    return;
        }
    }
    else if (listptr != argc)
            printusage();
        else
            switch (savearg) {
                case 2: exekute(argv[1],r1,r2,w1,w2);
                        return;
                case 4: exekute(argv[1],argv[3],r2,w1,w2);
                        return;
                case 5: exekute(argv[1],argv[3],argv[4],w1,w2);
                        return;
            }
}  /* end of getwfiles */

printusage()
{
    printf("Usage: pted inputfile [-r r1 [r2] ] [-w w1 [w2] ]\n");
    rmtmp();
}  /* end of printusage */
```

```
exekute(maintfile,read1,read2,write1,write2)
char *maintfile;
char *read1;
char *read2;
char *write1;
char *write2;
{
    char *command[300];

    concat(command,pted," ",maintfile," ",scr1," ",scr2," ",read1,
         " ",read2," ",write1," ",write2,0);

    if (system(command) < 0) {
        printf("pted: system call failed\n");
        rmtmp();
    }
}  /* end of exekute */
```

## Appendix 3: Test script.

A script for testing the features of pted follows; it is not a true path test, since that is not entirely possible from a script test of an editor, but all commands, pattern constructs, etc. occur somewhere in the script. It presumes the input file is at least 100 lines of text, and is somewhat inclined toward the text being some Pascal source code. Included in the script somewhere are commands that will reveal all four logic errors discovered in the Kernighan and Plauger version.

```
        { Pted text editor test script }
        { (assumes input file at least 100 lines long) }

        { line number tests }

77
1
100
50
0
-12
/var.*:/
//
??
?else begin?
$
58
.=
.
.+13
+13
.-7
/end/+5
.-3,.+3p
//;//;//;//
??,//p
6,7,8,9p

        { pattern matching tests }

/be/
/xrq3/
/...../
/[a-z][a-z]*.[a-z][a-z]*,[a-z][a-z]*/
/^/
/^    /
/^ *$/p
/^^/
/$$s/
/**/p
```

```
/[0-9][0-9]*/
/[^0-9]/
/n *b/p
/[^\^\/\\]/
/[^^]/
/[^\]]/p
/[-0-9-]/

      { backtracking, closures etc. }

67,78g/.*zz/p
57,66g/.*.*zz/p
a
aaaabbbbccccddddeeee
.
s/a*b*c*d*e*//p

      { commands }

a
hi there this is some splendidly boring text to
attempt to demonstrate the PT editor's append command.
.  (This line ok because there is more stuff after the '.').
.
.-2,.p
37a
this goes after line 37
.
38
22,30a
this goes after line 30 (the 22 is ignored).
.
31
a
.
0a
new line 1
.
1
$a
```

```
new last line
.
$
75,86p
75,86c
this replaces lines 75 to 86 incl.
.
75,86p
45
45c
you can change many lines to one, one to many, and
even many to many.
Great, eh?
.
57,59p
58dp
57,59p
600,700dp
29,41p
10,20,30,40dp
29,41p
w
1,$d
1,$p
r
50,60p
?.?i
This text will end up immediately before line 59;
it will therefore become line number 59.
.
59
1i
Inserted before line 1
.
1
0i
Inserted before line 0; effect is to wrap around and put it at the end.
.
$
66,73p
```

```
67,72m49
66,73p
49,58p
10,20m15
17,17m17
1
$
$m0
1
$
64
.,/end/p
1,15p
1,15g/^/m0
1,15p
p
.,.+7p
```

{ file I/O }

```
w
w 1
47,84w 2
/b/+6w 1
write
37,39p
37r 2
37,47p
1,5g/^/.r 1
1,20p
.w 2
r temp
0w 2
?..?r 2
.-6,.p
?[a-zA-Z]?,/[c-xC-X]/w 1
0r 1
1,8p
```

{ remaining non-global commands }

```
$-83a
junk to test substitute.........????????!"#$$$%&&'00(0)()(##$$$523e75
.
s/\.*//p
s/\.\.*//p
s/\?\?*/[&]/p
s/[0-9][0-9]*e[0-9]*$/$/p
s/$$$$$/ the end/p
s/&&/&&/p
s/(.*)/()/p
s/ //qp
s/\n//p
s/.*/&\n/p
s/sub/&\t/p
s/\t//p
s/j/^^^/p
s/^^^^//p
s/^/**/p
s/^**//p
s/**//p
s/st/\\t/p
s/\\t//p
s/[a-h]/x/gp
s:[st]:/:gp
s/.*//p
47,59s/[abc]/qlz/gp
.=
=
=p
?[a-z]$?=p
37=
46
=
$=p

      { global commands }

g/../a
v/zzzzzz/q
```

```
10,20g/./c
37,51g/a.*e/p
24,67g/begin/s//BEGIN/p
70,$g/^/m69
v/[a-m]/p
a
some text
.
.g/e/s///gp

    { and now, finally, the only command that can be }
    { demonstrated correctly only once:  }
     { after the file is written first }

w
37g
1,$g
/a/g
 g
```

BIBLIOGRAPHY:

(Kernighan)
Kernighan, B.W.; "A Tutorial Introduction to the UNIX Text Editor". Bell Laboratories Internal memorandum.

(Kernighan & Plauger76)
Kernighan, B.W. and Plauger, P.J.; "Software Tools", Addison-Wesley, Reading, Mass. 1976.

(Jensen & Wirth74)
Jensen, K. and Wirth, N.; "PASCAL User Manual and Report", Second Edition, Springer-Verlag, New York, 1974.

(Rosselet80)
Rosselet, J.A. "PT: a Pascal Subset", CSRG-119, Computer Systems Research Group, University of Toronto, 1980.

University of Toronto
Computer Systems Research Group

BIBLIOGRAPHY OF CSRG TECHNICAL REPORTS+

* CSRG-1 EMPIRICAL COMPARISON OF LR(k) AND PRECEDENCE PARSERS
J.J. Horning and W.R. Lalonde, September 1970
[ACM SIGPLAN Notices, November 1970]

* CSRG-2 AN EFFICIENT LALR PARSER GENERATOR
W.R. Lalonde, February 1971
[M.A.Sc. Thesis, EE 1971]

* CSRG-3 A PROCESSOR GENERATOR SYSTEM
J.D. Gorrie, February 1971
[M.A.Sc. Thesis, EE 1971]

* CSRG-4 DYLAN USER'S MANUAL
P.E. Bonzon, March 1971

CSRG-5 DIAL - A PROGRAMMING SYSTEM FOR INTERACTIVE ALGEBRAIC MANIPULATION
Alan C.M. Brown and J.J. Horning, March 1971

CSRG-6 ON DEADLOCK IN COMPUTER SYSTEMS
Richard C. Holt, April 1971
[Ph.D. Thesis, Dept. of Computer Science,
Cornell University, 1971]

CSRG-7 THE STAR-RING SYSTEM OF LOOSELY COUPLED DIGITAL DEVICES
John Neill Thomas Potvin, August 1971
[M.A.Sc. Thesis, EE 1971]

* CSRG-8 FILE ORGANIZATION AND STRUCTURE
G.M. Stacey, August 1971

CSRG-9 DESIGN STUDY FOR A TWO-DIMENSIONAL COMPUTER-ASSISTED ANIMATION SYSTEM
Kenneth B. Evans, January 1972
[M.Sc. Thesis, DCS, 1972]

* CSRG-10 HOW A PROGRAMMING LANGUAGE IS USED
William Gregg Alexander, February 1972
[M.Sc. Thesis, DCS 1971; Computer, v.8, n.11, November 1975]

* CSRG-11 PROJECT SUE STATUS REPORT
J.W. Atwood (ed.), April 1972

---

+ Abbreviations:
DCS - Department of Computer Science, University of Toronto
EE - Department of Electrical Engineering, University of Toronto
* - Out of print

* CSRG-12 THREE DIMENSIONAL DATA DISPLAY WITH HIDDEN LINE REMOVAL
Rupert Bramall, April 1972
[M.Sc. Thesis, DCS, 1971]

* CSRG-13 A SYNTAX DIRECTED ERROR RECOVERY METHOD
Lewis R. James, May 1972
[M.Sc. Thesis, DCS, 1972]

CSRG-14 THE USE OF SERVICE TIME DISTRIBUTIONS IN SCHEDULING
Kenneth C. Sevcik, May 1972
[Ph.D. Thesis, Committee on Information Sciences, University of Chicago, 1971; JACM, January 1974]

CSRG-15 PROCESS STRUCTURING
J.J. Horning and B. Randell, June 1972
[ACM Computing Surveys, March 1972]

CSRG-16 OPTIMAL PROCESSOR SCHEDULING WHEN SERVICE TIMES ARE HYPEREXPONENTIALLY DISTRIBUTED AND PREEMPTION OVERHEAD IS NOT NEGLIGIBLE
Kenneth C. Sevcik, June 1972
[Proceedings of the Symposium on Computer-Communication, Networks and Teletraffic, Polytechnic Institute of Brooklyn, 1972]

* CSRG-17 PROGRAMMING LANGUAGE TRANSLATION TECHNIQUES
W.M. McKeeman, July 1972

CSRG-18 A COMPARATIVE ANALYSIS OF SEVERAL DISK SCHEDULING ALGORITHMS
C.J.M. Turnbull, September 1972

CSRG-19 PROJECT SUE AS A LEARNING EXPERIENCE
K.C. Sevcik *et al*, September 1972
[Proceedings AFIPS Fall Joint Computer Conference, v. 41, December 1972]

* CSRG-20 A STUDY OF LANGUAGE DIRECTED COMPUTER DESIGN
David B. Wortman, December 1972
[Ph.D. Thesis, Computer Science Department, Stanford University, 1972]

CSRG-21 AN APL TERMINAL APPROACH TO COMPUTER MAPPING
R. Kvaternik, December 1972
[M.Sc. Thesis, DCS, 1972]

* CSRG-22 AN IMPLEMENTATION LANGUAGE FOR MINICOMPUTERS
G.G. Kalmar, January 1973
[M.Sc. Thesis, DCS, 1972]

CSRG-23 COMPILER STRUCTURE
W.M. McKeeman, January 1973
[Proceedings of the USA-Japan Computer Conference, 1972]

* CSRG-24 AN ANNOTATED BIBLIOGRAPHY ON COMPUTER PROGRAM ENGINEERING
J.D. Gannon (ed.), March 1973

CSRG-25 THE INVESTIGATION OF SERVICE TIME DISTRIBUTIONS
Eleanor A. Lester, April 1973
[M.Sc. Thesis, DCS, 1973]

* CSRG-26 PSYCHOLOGICAL COMPLEXITY OF COMPUTER PROGRAMS: AN INITIAL EXPERIMENT
Larry Weissman, August 1973

* CSRG-27 STRUCTURED SUBSETS OF THE PL/I LANGUAGE
Richard C. Holt and David B. Wortman, October 1973

* CSRG-28 ON REDUCED MATRIX REPRESENTATION OF LR(k) PARSER TABLES
Marc Louis Joliat, October 1973
[Ph.D. Thesis, EE 1973]

* CSRG-29 A STUDENT PROJECT FOR AN OPERATING SYSTEMS COURSE
B. Czarnik and D. Tsichritzis (eds.), November 1973

* CSRG-30 A PSEUDO-MACHINE FOR CODE GENERATION
Henry John Pasko, December 1973
[M.Sc. Thesis, DCS 1973]

* CSRG-31 AN ANNOTAED BIBLIOGRAPHY ON COMPUTER PROGRAM ENGINEERING
J.D. Gannon (ed.), Second Edition, March 1974

* CSRG-32 SCHEDULING MULTIPLE RESOURCE COMPUTER SYSTEMS
E.D. Lazowska, May 1974
[M.Sc. Thesis, DCS, 1974]

* CSRG-33 AN EDUCATIONAL DATA BASE MANAGEMENT SYSTEM
F. Lochovsky and D. Tsichritzis, May 1974
[INFOR, 14 (3), pp.270-278, 1976]

* CSRG-34 ALLOCATING STORAGE IN HIERARCHICAL DATA BASES
P. Bernstein and D. Tsichritzis, May 1974
[Information Systems Journal, v.1, pp.133-140]

* CSRG-35 ON IMPLEMENTATION OF RELATIONS
D. Tsichritzis, May 1974

* CSRG-36 SIX PL/I COMPILERS
D.B. Wortman, P.J. Khaiat, and D.M. Lasker, August 1974
[Software Practice and Experience, v.6, n.3, July-Sept. 1976]

* CSRG-37 A METHODOLOGY FOR STUDYING THE PSYCHOLOGICAL COMPLEXITY OF COMPUTER PROGRAMS
Laurence M. Weissman, August 1974
[Ph.D. Thesis, DCS, 1974]

* CSRG-38 AN INVESTIGATION OF A NEW METHOD OF CONSTRUCTING SOFTWARE
David M. Lasker, September 1974
[M.Sc. Thesis, DCS, 1974]

CSRG-39 AN ALGEBRAIC MODEL FOR STRING PATTERNS
Glenn F. Stewart, September 1974
[M.Sc. Thesis, DCS, 1974]

* CSRG-40 EDUCATIONAL DATA BASE SYSTEM USER'S MANUAL
J. Klebanoff, F. Lochovsky, A. Rozitis, and
D. Tsichritzis, September 1974

* CSRG-41 NOTES FROM A WORKSHOP ON THE ATTAINMENT OF RELIABLE SOFTWARE
David B. Wortman (ed.), September 1974

* CSRG-42 THE PROJECT SUE SYSTEM LANGUAGE REFERENCE MANUAL
B.L. Clark and F.J.B. Ham, September 1974

* CSRG-43 A DATA BASE PROCESSOR
E.A. Ozkarahan, S.A. Schuster and K.C. Smith,
November 1974 [Proceedings National Computer Conference 1975, v.44, pp.379-388]

* CSRG-44 MATCHING PROGRAM AND DATA REPRESENTATION TO A COMPUTING ENVIRONMENT
Eric C.R. Hehner, Novemver 1974
[Ph.D. Thesis, DCS, 1974]
See Computer, Vol.9, No.9, August 1976, pp.65-70.

* CSRG-45 THREE APPROACHES TO RELIABLE SOFTWARE; LANGUAGE DESIGN, DYADIC SPECIFICATIONS, COMPLEMENTARY SEMANTICS
J.E. Donahue, J.D. Gannon, J.V. Guttag and
J.J. Horning, December 1974

CSRG-46 THE SYNTHESIS OF OPTIMAL DECISION TREES FROM DECISION TABLES
Helmut Schumacher, December 1974
[M.Sc. Thesis, DCS, 1974; CACM, v.19, n.6, June 1976]

* CSRG-47 LANGUAGE DESIGN TO ENHANCE PROGRAMMING RELIABILITY
John D. Gannon, January 1975
[Ph.D. Thesis, DCS, 1975]

* CSRG-48 DETERMINISTIC LEFT TO RIGHT PARSING
Christopher J.M. Turnbull, January 1975
[Ph.D. Thesis, EE, 1974]

* CSRG-49 A NETWORK FRAMEWORK FOR RELATIONAL IMPLEMENTATION
D. Tsichritzis, February 1975 [in Data Base Description, Dongue and Nijssen (eds.), North Holland Publishing Co.]

* CSRG-50 A UNIFIED APPROACH TO FUNCTIONAL DEPENDENCIES AND RELATIONS
P.A. Bernstein, J.R. Swenson and D.C. Tsichritzis
February 1975 [Proceedings of the ACM SIGMOD Conference, 1975]

* CSRG-51 ZETA: A PROTOTYPE RELATIONAL DATA BASE MANAGEMENT SYSTEM
M. Brodie (ed). February 1975 [Proceedings Pacific ACM Conference, 1975]

* CSRG-52 AUTOMATIC GENERATION OF SYNTAX-REPAIRING AND PARAGRAPHING PARSERS
David T. Barnard, March 1975
[M.Sc. Thesis, DCS, 1975]

* CSRG-53 QUERY EXECUTION AND INDEX SELECTION FOR RELATIONAL DATA BASES
J.H. Gilles Farley and Stewart A. Schuster, March 1975

CSRG-54 AN ANNOTATED BIBLIOGRAPHY ON COMPUTER PROGRAM ENGINEERING
J.V. Guttag (ed.), Third Edition, April 1975

CSRG-55 STRUCTURED SUBSETS OF THE PL/1 LANGUAGE
Richard C. Holt and David B. Wortman, May 1975

* CSRG-56 FEATURES OF A CONCEPTUAL SCHEMA
D. Tsichritzis, June 1975 [Proceedings Very Large Data Base Conference, 1975]

* CSRG-57 MERLIN: TOWARDS AN IDEAL PROGRAMMING LANGUAGE
Eric C.R. Hehner, July 1975
see Acta Informatica Col.10, No.3, pp.229-243, 1978

CSRG-58 ON THE SEMANTICS OF THE RELATIONAL DATA MODEL
Hans Albrecht Schmid and J. Richard Swenson,
July 1975 [Proceedings of the ACM SIGMOD Conference, 1975]

* CSRG-59 THE SPECIFICATION AND APPLICATION TO PROGRAMMING OF ABSTRACT DATA TYPES
John V. Guttag, September 1975
[Ph.D. Thesis, DCS, 1975]

* CSRG-60 NORMALIZATION AND FUNCTIONAL DEPENDENCIES IN THE RELATIONAL DATA BASE MODEL
Phillip Alan Bernstein, October 1975
[Ph.D. Thesis, DCS, 1975]

* CSRG-61 LSL: A LINK AND SELECTION LANGUAGE
D. Tsichritzis, November 1975 [Proceedings ACM SIGMOD Conference, 1976]

* CSRG-62 COMPLEMENTARY DEFINITIONS OF PROGRAMMING LANGUAGE SEMANTICS
James E. Donahue, November 1975
[Ph.D. Thesis, DCS, 1975]

CSRG-63 AN EXPERIMENTAL EVALUATION OF CHESS PLAYING HEURISTICS
Lazlo Sugar, December 1975
[M.Sc. Thesis, DCS, 1975]

CSRG-64 A VIRTUAL MEMORY SYSTEM FOR A RELATIONAL ASSOCIATIVE PROCESSOR
S.A. Schuster, E.A. Ozkarahan, and K.C. Smith,
February 1976 [Proceedings National Computer Conference 1976, v.45, pp.855-862]

CSRG-65 PERFORMANCE EVALUATION OF A RELATIONAL ASSOCIATIVE PROCESSOR
E.A. Ozkarahan, S.A. Schuster, and K.C. Sevcik,
February 1976 [ACM Transactions on Database Systems, v.1, n:4, December 1976]

CSRG-66 EDITING COMPUTER ANIMATED FILM
Michael D. Tilson, February 1976
[M.Sc. Thesis, DCS, 1975]

CSRG-67 A DIAGRAMMATIC APPROACH TO PROGRAMMING LANGUAGE SEMANTICS
James R. Cordy, March 1976
[M.Sc. Thesis, DCS, 1976]

* CSRG-68 A SYNTHETIC ENGLISH QUERY LANGUAGE FOR A RELATIONAL ASSOCIATIVE PROCESSOR
L. Kerschberg, E.A. Ozkarahan, and J.E.S. Pacheco,
April 1976

CSRG-69 AN ANNOTATED BIBLIOGRAPHY ON COMPUTER PROGRAM ENCINEERING
D. Barnard and D. Thompson (eds.), Fourth Edition,
May 1976

* CSRG-70 A TAXONOMY OF DATA MODELS
L. Kerschberg, A. Klug, and D.Tsichritzis, May 1976
[Proceedings Very Large Data Base Conference, 1976]

* CSRG-71 OPTIMIZATION FEATURES FOR THE ARCHITECTURE OF A DATA BASE MACHINE
E.A. Ozkarahan and K.C. Sevcik, May 1976
[ACM Transactions of Database Systems, v.2, n.4, December 1977]

* CSRG-72 THE RELATIONAL DATA BASE SYSTEM OMEGA - PROGRESS REPORT
H.A. Schmid (ed.), P.A. Bernstein (ed.), B. Arlow,
R. Baker and S. Pozgaj, July 1976

CSRG-73 AN ALGORITHMIC APPROACH TO NORMALIZATION OF RELATIONAL DATA BASE SCHEMAS
P.A. Bernstein and C. Beeri, September 1976

* CSRG-74 A HIGH-LEVEL MACHINE-ORIENTED ASSEMBLER LANGUAGE FOR A DATA BASE MACHINE
E.A. Ozkarahan and S.A. Schuster, October 1976

CSRG-75 DO CONSIDERED OD: A CONTRIBUTION TO THE PROGRAMMING CALCULUS
Eric C.R. Hehner, November 1976
Acta Informatica to appear 1979

CSRG-76 SOFTWARE HUT: A COMPUTER PROGRAM ENGINEERING PROJECT IN THE FORM OF A GAME
J.J. Horning and D.B. Wortman, November 1976
[IEEE Transactions on Software Engineering, v.SE-3, n.4, July 1977]

CSRG-77 A SHORT STUDY OF PROGRAM AND MEMORY POLICY BEHAVIOUR
G. Scott Graham, January 1977

CSRG-78 A PANACHE OF DBMS IDEAS
D. Tsichritzis (ed.), February 1977

CSRG-79 THE DESIGN AND IMPLEMENTATION OF AN ADVANCED LALR PARSE TABLE CONSTRUCTOR
David H. Thompson, April 1977
[M.Sc. Thesis, DCS, 1976]

CSRG-80 AN ANNOTATED BIBLIOGRAPHY ON COMPUTER PROGRAM ENGINEERING
D. Barnard (ed.), Fifth Edition, May 1977

CSRG-81 PROGRAMMING METHODOLOGY: AN ANNOTATED BIBLIOGRAPHY FOR IFIP WORKING GROUP 2.3
Sol J. Greenspan and J.J. Horning (eds.), First Edition, May 1977

CSRG-82 NOTES ON EUCLID
edited by W. David Elliot and David T. Barnard, August 1977

CSRG-83 TOPICS IN QUEUEING NETWORK MODELING
edited by G. Scott Graham, July 1977

CSRG-84 TOWARD PROGRAM ILLUSTRATION
Edward Yarwood, September 1977
[M.Sc. Thesis, DCS, 1974]

CSRG-85 CHARACTERIZING SERVICE TIME AND RESPONSE TIME DISTRIBUTIONS IN QUEUEING NETWORK MODELS OF COMPUTER SYSTEMS
Edward D. Lazowska, September 1977
[Ph.D. Thesis, DCS, 1977]

CSRG-86 MEASUREMENTS OF COMPUTER SYSTEMS FOR QUEUEING NETWORK MODELS
Martin G. Kienzle, October 1977
[M.Sc. Thesis, DCS, 1977; Proc. Int. Symp. on Modelling and Performance Evaluation of Computer Systems, Vienna, 1979]

CSRG-87 'OLGA' LANGUAGE REFERENCE MANUAL
B. Abourbih, H. Trickey, D.M. Lewis, E.S. Lee, P.I.P. Boulton, November 1977

CSRG-88 USING A GRAMMATICAL FORMALISM AS A PROGRAMMING LANGUAGE
Brad A. Silverberg, January 1978
[M.Sc. Thesis, DCS, 1978]

CSRG-89 ON THE IMPLEMENTATION OF RELATIONS: A KEY TO EFFICIENCY
Joachim W. Schmidt, January 1978

CSRG-90 DATA BASE MANAGEMENT SYSTEM USER PERFORMANCE
Frederick H. Lochovsky, April 1978
[Ph.D. Thesis, DCS, 1978]

CSRG-91 SPECIFICATION AND VERIFICATION OF DATA BASE SEMANTIC INTEGRITY
Michael Lawrence Brodie, April 1978
[Ph.D. Thesis, DCS, 1978]

CSRG-92 STRUCTURED SOUND SYNTHESIS PROJECT (SSSP): AN INTRODUCTION
by William Buxton, Guy Fedorkow, with Ronald Baecker, Gustav Ciamaga, Leslie Mezei and K.C. Smith, June 1978

CSRG-93 A DEVICE-INDEPENDENT,GENERAL-PURPOSE GRAPHICS SYSTEM IN A MINICOMPUTER TIME-SHARING ENVIRONMENT
William T. Reeves, August 1978
[M.Sc. Thesis, DCS, 1976]

CSRG-94 ON THE AXIOMATIC VERIFICATION OF CONCURRENT ALGORITHMS
Christian Lengauer, August 1978
[M.Sc. Thesis, DCS, 1978]

CSRG-95 PISA: A PROGRAMMING SYSTEM FOR INTERACTIVE PRODUCTION OF APPLICATION SOFTWARE
Rudolf Marty, August 1978

CSRG-96 ADAPTIVE MICROPROGRAMMING AND PROCESSOR MODELING
Walter G. Rosocha
[Ph.D. Thesis, EE, August 1978]

CSRG-97 DESIGN ISSUES IN THE FOUNDATION OF A COMPUTER-BASED TOOL FOR MUSIC COMPOSITION
William Buxton
[M.Sc. Thesis, CSRG, October 1978]

CSRG-98 THEORY OF DATABASE MAPPINGS
Anthony C. Klug
[Ph.D. Thesis, DCS, December 1978]

CSRG-99 HIERARCHICAL COROUTINES: A MECHANISM FOR IMPROVED PROGRAM STRUCTURE
Leonard I. Vanek, February 1979

CSRG-100 TOPICS IN PERFORMANCE EVALUATION
G. Scott Graham (ed.), July 1979

CSRG-101 A PANACHE OF DBMS IDEAS II
F.H. Lochovsky (ed.), May 1979

CSRG-102 A SIMPLE SET THEORY FOR COMPUTING SCIENCE
Eric C.R. Hehner, May 1979

CSRG-103 THE CENTRALIZED ALGORITHM IN DISTRIBUTED SYSTEMS
Ernest J.H. Chang
[Ph.D. Thesis, DCS, July 1979]

CSRG-104 ELIMINATING THE VARIABLE FROM DIJKSTRA'S MINI-LANGUAGE
D. Hugh Redelmeier, July 1979

CSRG-105 A LANGUAGE FACILITY FOR DESIGNING INTERACTIVE DATABASE-INTENSIVE APPLICATIONS
John Mylopoulos, Philip A. Bernstein, Harry K.T. Wong, July 1979

CSRG-106 ON APPROXIMATE SOLUTION TECHNIQUES FOR QUEUEING NETWORK MODELS OF COMPUTER SYSTEMS
Satish Kumar Tripathi, July 1979

CSRG-107 A FRAMEWORK FOR VISUAL MOTION UNDERSTANDING
John K. Tsotsos, John Mylopoulos, H. Dominic Covvey Steven W. Zucker, DCS, June 1979

CSRG-108 DIALOGUE ORGANIZATION AND STRUCTURE FOR INTERACTIVE INFORMATION SYSTEMS
John Leonard Barron
[M.Sc. Thesis, DCS, 1980]

CSRG-109 A UNIFYING MODEL OF PHYSICAL DATABASES
D.S. Batory, C.C. Gotlieb, April 1980

CSRG-110 OPTIMAL FILE DESIGNS AND REORGANIZATION POINTS
D.S. Batory, April 1980

CSRG-111 A PANACHE OF DBMS IDEAS III
D. Tsichritzis (ed.), April 1980

CSRG-112 TOPICS IN PSN - II: EXCEPTIONAL CONDITION HANDLING IN PSN; REPRESENTING PROGRAMS IN PSN; CONTENTS IN PSN
Yves Lesperance, Byran M. Kramer, Peter F. Schneider
April, 1980.

CSRG-113 SYSTEM-ORIENTED MACRO-SCHEDULING
C.C. Gotlieb and A. Schonbach
May 1980

CSRG-114 A FRAMEWORK FOR VISUAL MOTION UNDERSTANDING
John Konstantine Tsotsos
[Ph.D. Thesis, DCS, June 1980]

CSRG-115 SPECIFICATION OF CONCURRENT EUCLID
James R. Cordy and Richard C. Holt
July 1980

CSRG-116 THE REPRESENTATION OF PROGRAMS IN THE PROCEDURAL SEMANTIC NETWORK FORMALISM
Bryan M. Kramer
[M.Sc. Thesis, DCS, 1980]

CSRG-117 CONTEXT-FREE GRAMMARS AND DERIVATION TREES AS PROGRAMMING TOOLS
Volker Linnemann
September 1980

CSRG-118 S/SL: SYNTAX/SEMANTIC LANGUAGE INTRODUCTION AND SPECIFICATION
R.C. Holt, J.R. Cordy, D.B. Wortman
CSRG, September 1980

CSRG-119 PT: A PASCAL SUBSET
Alan Rosselet
[M.Sc. Thesis, DCS, October 1980]

CSRG-120 PTED: A STANDARD PASCAL TEXT EDITOR BASED ON THE KERNIGHAN AND PLAUGER DESIGN
Ken Newman, DCS
October 1980